Test Selection Practices in a Large IT Company

Vincent Blondeau

Université des Sciences et Technologies de Lille – Lille 1

Département de formation doctorale en informatique **École doctorale SPI Lille**
UFR IEEA

Test Selection Practices in a Large IT Company

THÈSE

présentée et soutenue publiquement le 8 Novembre 2017

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille

(spécialité informatique)

par

Vincent Blondeau

Composition du jury

Rapporteurs :	Anthony Clève (Professor – Université de Namur) Andy Zaidman (Associate Professor – Delft University of Technology)
Examinateurs :	Serge Demeyer (Professor – University of Antwerp) Clarisse Dhaenens (Professor – Université Lille 1) Henda Hajjami Ben Ghezala (Professor – ENSI)
Directeur de thèse :	Nicolas Anquetil (Associate Professor – Université Lille 1)
Co-Encadrante de thèse :	Anne Etien (Associate Professor – Université Lille 1)
Invités :	Sylvain Cresson (Worldline) Pascal Croisy (Worldline)

Centre de Recherche en Informatique, Signal et Automatique de Lille — UMR USTL/CNRS 9189
INRIA Lille - Nord Europe

Numéro d'ordre: 42429

Acknowledgments

I would like to thank my thesis supervisors Nicolas Anquetil and Anne Etien for allowing me to do a Ph.D at the RMoD group, as well as helping and supporting me during the three years of my Ph.D. I thank also my managers at Worldline, Sylvain Cresson and Pascal Croisy for their support and help. I thank the thesis reviewers and jury members Antony Clève, Andy Zaidman for kindly reviewing my thesis and providing me valuable feedback. I thank the jury members Serge Demayer, Clarisse Dhaenens, and Henda Hajjami Ben Ghezala. Many thanks to the RMoD group members, the SDCO transversal business unit members and everyone that helped me to achieve my works. I personally wanted to thank Olivier Auverlot, Clément Bera, Julien Delplanque, Stéphane Ducasse, Cyril Ferlicot, Guillaume Larchevêque, Brice Govin, Thibault Raffaillac, Gustavo Santos, and Benoît Verhaeghe for their collaboration and fellowship. Finally, I thank my family and friends for the support during this adventure.

Abstract

Nowadays, Worldline, a major IT company, develops application that are dealing with an increasing amount of critical data with a source code is more and more complex. To the stay in the race against its competitors, Worldline has to increase the quality of its projects. One transversal team of the company has for main mission to enhance project quality. By joining this team, we performed an audit of several projects of the company to identify how to avoid project failure. Relying on project metadata analysis, interviews, survey, and literature study, this audit drew no final conclusion. However, it highlighted a need to enhance testing usage.

To test every piece of code is compulsory to avoid project failure. In industry and more specifically in Worldline, automation of tests has begun to ensure the proper behavior of their applications. But, ensuring the execution of all tests after a change can be a costly operation requiring several hours. However, in a daily development process, developers can not run all the tests after a change to check the impact of their modifications. Feedback on the changed code is delayed and developer have to spent time to discover the potential bugs. A late feedback can potentially lead to the failure of a project.

The solution generally proposed in literature consists in reducing the number of tests to run by selecting only the ones related to the last changes made by the developer. The approach selects a suitable and small set of tests to detect potential regression in the application behavior.

Test selection approaches have their own advantages and drawbacks. These approaches need to be studied in the context of Worldline and consequently adapted to suit developers habits. Before integrating such an adapted test selection approach, a study of the testing behavior of the developers have been done to get their current test selection usage. This study will monitor all the tests launched by the developers during their everyday development tasks.

Based on the study of the Worldline environment, we developed a tool for the developers aiming to perform test selection. It is adapted to the developers habits and is directly usable by them. The goal is also to study the impact of this tool on their behavior. We hope that the test selection approach that is packaged in a tool adapted to the Worldline developers will bring a change in their development process.

Keywords: Software Maintenance, Testing, Automated Test Selection, Industrial

Résumé

De nos jours, Worldline, une importante société d'informatique, développe des applications qui traitent un nombre croissant de données critiques avec un code source de plus en plus complexe. Pour rester dans la course contre ses concurrents, Worldline doit améliorer la qualité de ses projets. Une équipe transversale de la société a pour mission principale d'améliorer la qualité des projets. En rejoignant cette équipe, nous avons réalisé un audit sur plusieurs projets de l'entreprise afin d'identifier comment éviter l'échec des projets. En se basant sur une analyse de métadonnées, des interviews, des sondages, et une étude de la littérature, cet audit n'a finalement tiré aucune conclusion. Cependant, il a mis en lumière le besoin d'améliorer l'utilisation des tests.

Effectivement, tester chaque partie du code est requis pour éviter l'échec du projet. Dans l'industrie, et plus spécifiquement à Worldline, l'automatisation des tests a commencé dans le but de maîtriser le comportement des applications. Mais, s'assurer de l'exécution de tous les tests après un changement peut être une opération coûteuse requérant plusieurs heures. Le retour sur le code changé est retardé et le développeur perd du temps pour découvrir les potentiels bogues. Ainsi, un retour tardif peut potentiellement amener le projet à l'échec.

La solution généralement proposée dans la littérature consiste à réduire le nombre de tests à lancer en sélectionnant seulement ceux relatifs aux derniers changements effectués par le développeur. L'approche sélectionne un jeu de tests réduit et approprié qui détectera de potentielles régressions dans le comportement de l'application.

Chaque approche de sélection de tests a ses propres avantages et inconvénients. Elles ont donc besoin d'être étudiées dans le contexte de Worldline et adaptées en conséquence pour convenir aux habitudes des développeurs. Avant d'intégrer une telle approche, une étude a été faite pour connaître les habitudes actuelles des développeurs vis-à-vis de la sélection de tests. Cette étude surveille tous les tests lancés par les développeurs pendant leurs tâches quotidiennes de développement.

Grâce à l'étude de l'environnement de Worldline, nous avons développé un outil pour les développeurs visant à effectuer la sélection des tests. Il est adapté aux habitudes des développeurs et leur est directement utilisable. Le but est aussi d'étudier l'impact de cet outil sur leur comportement. Nous espérons que cette approche de sélection de tests ainsi contenue dans un outil adapté aux développeurs de Worldline, apportera des changements dans leur processus de développement.

Mots-clés: Maintenance Logicielle, Tests, Sélection Automatisée de Tests, Industrie

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Contents

1 Context

Competition between IT companies is tough. Each company wants to attract client projects, implement, and maintain them. But, this has a cost: to ensure the success of a project, the employees have to ensure that the client has the requested application in time, within budget, and with all the desired features implemented. Nowadays, applications of Worldline, a major IT company, are dealing with an increasing amount of critical data, and their source code is more and more complex. To stay in the race, the company has to reinvent itself continually.

One transversal team of the company has for mission to provide tools, expertise and support to the development teams. The goal of this team is to ease the day to day work of developers in particular, and enhance project quality in general. By joining this team, we performed an audit of several projects of the company to try to identify how to avoid project failure. This audit, described in the first chapter of this thesis, relies on project metadata analysis, interviews, survey, and literature study. The root causes of project failure identified mainly point the lack of communication between the client and the project team. However, as a byproduct, the audit also highlighted a need to enhance testing practice.

To avoid failure, testing every piece of code is compulsory. But, for a long time, applications had to be tested manually: no support was given to the developers to test complex application automatically. Despite the fact that this practice

still exists, developers began, in industry Bertolino [2007], and more specifically at Worldline, to automate tests to ensure the proper behavior of their applications. But, at Worldline, these tests take too much time to be executed frequently. Having yet no knowledge on how testing is practiced in the company, we hypothesized that this execution often happens only at night thanks to continuous integration tooling. Thus, tests do not provide immediate feedback; they fail in one of their main missions. To convince the transversal team of imposing a change in development practices of thousands of developers, we needed hard data on the pros and cons of the impact of launching tests sooner to give developers faster feedback.

2 Problem

At Worldline, ensuring the execution of all tests after a change can be a costly operation requiring several hours. However, in a daily development process, developers can not run all the tests after a change to check the impact of their modifications. This operation may require installing, configuring and updating databases or others environments as well as testing abnormal running conditions such as timeout on server connection. They do not have time to do that due to stress and deadlines in the project. Consequently, we observed that developers very often skip tests during the day and run them only at night thanks to continuous integration servers. But, that is not a viable solution. Feedback on the changed code should be fast to help identify the potential bugs more quickly. Delayed feedback harms software development and can potentially lead to the failure of a project.

In this manuscript, we focus on two kinds of automated tests: customer tests and developer tests (based on the test taxonomy of Meszaros [2007]).

- *"A customer test verifies the behavior of a slice of the visible functionality of the overall system. The system under test may consist of the entire system or a fully functional top-to-bottom slice of the system".*
- *"An unit test is a test that verifies some small parts of the overall system. What turns a test into a unit test is that the system under test is a very small subset of the overall system and may be unrecognizable to someone who is not involved in building the software. In contrast to a customer test, which is derived almost entirely from the requirements and which should be verifiable by the customer."*

These two kinds of tests can be found at Worldline. However, in case of time pressure, the customer tests (checking customer requirements) take priority and the unit tests are left apart. To avoid this, some best practices are defined at Worldline. They state that the developers should:

- Cover at least 80% of source code with automated tests.
- Use Continuous Integration to compile and test their application.

- Follow a "git workflow", *i.e.,* use branches and pull requests to integrate their features.

However, a great deal of freedom is also given to the developers and project leaders to choose their own technology stack, development technique, strategy, etc. Thus for example, agile methods or Test Driven Development are progressively used inside the company but remain marginal at the moment of this writing. Worldline being a huge company, no deep analysis has been performed to identify the projects following these best practices or not. In our study of different projects, we found that they were often laid aside.

A solution generally proposed in literature to get faster feedback on the changed code consists in reducing the number of tests to run by selecting only the ones related to the last changes made by the developer. The approach selects a suitable and small set of tests to detect potential regression in the application behavior. We expect that such a test selection approach could change the habits of the Worldline developers. Two consequences of this change are foreseen: first, they will not relaunch all the tests of the application each time; second, they will be encouraged to run tests more often.

Among test selection approaches, two suit our needs: static and dynamic. The static approach creates a model of the source code and explores it to find links between changed methods and tests. The dynamic approach records invocations of methods during the execution of test scenarios. These two approaches have their own advantages and drawbacks that need to be studied in the context of Worldline and consequently adapted. Moreover, to convince the company of the impact of such approaches on the work of its developers, we need, first, to study their habits before using it and, second, to compare them with their new practices, *i.e.,* adopting an adapted test selection approach.

However, to be relevant, this study has to be performed on as many developers as possible, for the longest period possible,... Also, these studies have to be performed without extra cost for the developers, *i.e.,* included in their development environment and not delay any of their activities.

3 Our Approach in a Nutshell

In our goal to change developers habits by introducing test selection, our first action is to shape a test selection approach adapted to the Worldline environment. Worldline is the European leader in the payments and transactional services industry. Its applications have to take care of a substantial amount of financial transactions. A plethora of libraries and frameworks are used. It can impede some test selection approaches to operate. Furthermore, the Java programming language is mainly used. Despite this language being statically typed, thus facilitating a static

approach, some particularities of the language like inheritance or reflexivity, could be a problem for test selection (see Chapter 4). So, these specificities have to be considered to propose an adapted test selection solution.

Another constraint, due to the industrial context of this study, is that our experiments should be perceived by developers as being transparent, and certainly not as disturbing their usual work. A study of the testing behavior of the developers is mandatory to get their current test selection usage before integrating such a test selection approach. This study will monitor all the tests launched by the developers during their everyday development tasks.

Based on the study of the Worldline environment from a technical point of view, we will develop an adapted tool. It will take into account the identified issues and will be directly usable by the developers. The underlying goal is also to study the impact of this tool on their behavior. We hope that the test selection approach that is packaged in a tool adapted to the Worldline developers will bring a change in their development process.

So, the research question of the thesis is: Can we improve the testing habits of Worldline developers by giving them faster feedback on their source code modifications?

The hidden assumption behind this research question is that by improving testing habits, we will ultimately improve the chances of success of the projects. However, this assumption will not be formally tested because of time constraints.

4 Contributions

The main contributions of this thesis are:

- An audit of the developers awareness about project success and failure and root causes of software failure at Worldline.
- A classification of problems that may arise when trying to identify the tests that cover a method. We give concrete examples of these problems and list some possible solutions.
- A field study on how Worldline developers use tests in their daily practice, whether they use tests selection and why they do. Results are reinforced by interviews with developers involved in the study.
- A study of the usage of our test selection tool on the developers.

5 Structure of the Dissertation

The structure of the dissertation is:

- Chapter 2 describes the motivation of the dissertation: it studies the ability of metrics to predict project success and gather the insights of the developers

about project health.
- Chapter 3 gives the state of the art about test selection approaches and the study of developers habits.
- Chapter 4 presents issues of the test selection approaches, the methodologies to resolve them, and the impact of their resolution on the test selection approaches.
- Chapter 5 describes the experiment on testing habits of Worldline's developers and gives results on their usage.
- Chapter 6 evaluates the usage of a test selection tool on the Worldline developers through data analysis and interviews.
- Chapter 7 concludes and gives perspectives for future works.

6 List of Publications

This is the complete list of all our publications in chronological order:

i. Vincent Blondeau, Nicolas Anquetil, Stéphane Ducasse, Sylvain Cresson, and Pascal Croisy. Software metrics to predict the health of a project? In *IWST'15*, Brescia, Italy, July 2015a. doi: 10.1145/2811237.2811294. URL http://rmod.inria.fr/archives/papers/Blon15a-IWST-Software_metrics_to_predict_the_health_of_a_project.pdf

ii. Vincent Blondeau, Sylvain Cresson, Pascal Croisy, Anne Etien, Nicolas Anquetil, and Stéphane Ducasse. Predicting the Health of a Project? An Assessment in a Major IT Company. In *SATToSE'15*, Mons, Belgium, July 2015b. URL http://rmod.inria.fr/archives/papers/Blon15b-SATToSE-Predicting_the_Health_of_a_Project-An_Assessment_in_a_Major_IT_company.pdf

iii. Vincent Blondeau, Sylvain Cresson, Pascal Croisy, Anne Etien, Nicolas Anquetil, and Stéphane Ducasse. Dynamic and Static Approaches Comparison for Test Suite Reduction in Industry. In *BENEVOL'15: 14th BElgian-NEtherlands software eVOLution seminar*, Lille, France, December 2015c. URL http://rmod.inria.fr/archives/papers/Blon15c-BENEVOL-Blondeau_Vincent-Dynamic_and_Static_Approaches_Comparison_for_Test_Suite_Reduction_in_Industry.pdf

iv. Vincent Blondeau, Anne Etien, Nicolas Anquetil, Sylvain Cresson, Pascal Croisy, and Stéphane Ducasse. Test case selection in industry: An analysis of issues related to static approaches. *Software Quality Journal*, pages 1–35, 2016b. ISSN 1573-1367. doi: 10.1007/s11219-016-9328-4. URL http://rmod.inria.fr/archives/papers/Blon16a-Software_Quality_Journal-Test_Case_Selection_in_Industry-An_Analysis_of_Issues_Related_to_Static_Approaches.pdf

v. Vincent Blondeau, Nicolas Anquetil, Stéphane Ducasse, Sylvain Cresson, and Pascal Croisy. Test selection with moose in industry: Impact of granularity. In *International Workshop on Smalltalk Technologies IWST'16*, Prague, Czech Republic, September 2016a. URL http://rmod.inria.fr/archives/papers/Blon16b-IWST-Test_Selection_with_Moose_In_Industry.pdf

vi. Benoit Verhaeghe, Nicolas Anquetil, Stéphane Ducasse, and Vincent Blondeau. Usage of tests in an open-source community. In *IWST'17*, Maribor, Slovenia, September 2017. URL https://hal.inria.fr/hal-01579106

vii. Vincent Blondeau, Anne Etien, Nicolas Anquetil, Sylvain Cresson, Pascal Croisy, and Stéphane Ducasse. What are the Testing Habits of Developers? A Case Study in a Large IT Company. In *Proceedings of the 21st IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*, Shanghai, China, August 2017. URL https://hal.inria.fr/hal-01571655/file/ICSME-FinalVersion-PDF_eXpress-Certified.pdf

CHAPTER 2

Motivation

Contents

Despite testing being the main topic of my thesis, it was not the first. We began to focus on using big data to predict the failure of Worldline development projects.

1 Predicting the Health of a Project

Project teams create a lot of data: process metrics, bug reports, source code, continuous integration artefacts, production metrics, social network communications, etc. On the other hand, big data approaches are becoming predominant in companies. Worldline would like to take advantage of data science, and especially statistics, to help them evaluate the health of their projects. To monitor projects health pro-actively, they would like to mine past projects data, *e.g.,* metrics on software, bugs, budget, production issues, performance, team communications,. . . , and provide alerts to take action on the project. The hope is that some software metrics could be tracked to predict failure risks or confirm good health. If a factor of success is found, managers could anticipate projects failures and take early actions to help or to monitor closely the project, allowing one to act in a preventive mode rather than a corrective one. Finding the right metrics in the whole data set is challenging and applying it on a given project ahead of time even more.

For IT projects management, two main approaches are used at Worldline: the Agile and the Waterfall processes. Agile development is still marginal: little data is available. A Waterfall process is more frequently used in Worldline, this will be the case for all the projects analyzed in this chapter. When the waterfall process is used, Worldline assesses project health according to three items: resources, schedule, and scope. *Resources*, are measured in $man\text{-}days$. They are provisioned at the beginning of the project. A project that does not respect the allocated resources has a decreasing health and is said to be in slippage. The *slippage* is the number of $man\text{-}days$ a project uses in excess to the initially provisioned resources. The

schedule is defined with the client as a series of milestones. An example of milestone can be a release of the application to the client. These milestones should be respected to avoid a decrease of the project health. The *scope* of the application is defined at the beginning of the project with the client. If a project does not deliver the required features at a given milestone, project health decreases too. Thereby, if the project respects the initial provisioned resources and schedule, and, provides the features required by the client, then the project is in good health. Otherwise, if one of the three items is not respected, the project is in bad health. But the health of the project is only an assessment during the project without possibility to anticipate it.

To find relationship inside the available data, we reviewed literature on project success predicted by data mining. Second, we experimented with more than 10 project metrics on 50 real world, closed-source, projects of Worldline to find correlation. Third, we realized interviews with project managers to find indicators that could be linked to project health.

1.1 Systematic Literature Review

1.1.1 Methodology

To identify previous work that could have already proposed relationships between project health and some project metrics, we reviewed the literature. As defined by the guidelines of Keele et al. [2007], we first stated the following research question: *"What metrics could help predict the success or failure of a project?"* and defined a structure for the research questions:
Population: Projects
Intervention: Predict success or failure with metrics
Comparison: Cross projects
Outcome: Metrics used
Context: Industrial
From this structure, we defined the query to use in scientific articles databases:

"Predict AND Project AND metric AND software AND (success OR failure)"

We searched in the main data sources for our domain (software engineering): *IEEEXplore*, *ACM*, *Science Direct*, *Springer Link*, and *Wiley*. For each data source, we searched the keywords in the meta data.

The results then need to be filtered to select only the papers related to the research question. We performed a three step process to select the relevant papers: the first selection was on the title only, the second one was on abstract, and the last one was on the full text of the article. Moreover, we included only the papers in English and peer-reviewed, and, excluded the short papers and papers that do

not study real projects. For one paper, we were not able to access its full text, and discarded it from the study. Table 2.1 summarizes the count of the papers we obtained.

Table 2.1: Description of the Systematic Literature Review

Data source	Results	Selection on		
		Title	Abstract	Full Text
IEEExplore	7	3	2	0
ACM	189	52	11	4
Science Direct	128	9	4	3
Springer Link	1370	45	15	7
Wiley	2	2	2	0
Total	1696	111	34	14

For the 14 selected papers, we answered these questions:

- What is the title of the paper?
- Who are the authors?
- Is the paper a Conference or Journal paper?
- What is the kind of the project (open source, company, or students)?
- What is the company name (if applicable)?
- How many projects are implied in the study?
- What are the used techniques to predict the success or the failure of the project?
- What are the metrics used for the prediction?
- Is a comparison made with other prediction models? Which ones?
- What is the accuracy of the approach?
- What are the results? (*i.e.,* the metrics that predict the success or failure)
- What are the conclusions?

This helped us organize the papers and report them.

1.1.2 Results

Fitzgerald et al. [2012] studied seven open-source projects to predict project failure. They first analyzed the discussion between project participants and extract metrics. Then, they used them as input of classification algorithms. The authors conclude that building early prediction model can provide a positive value to a project.

Fukushima et al. [2014] built predictive models with 11 open source projects. They studied within project and cross project predictions thanks to 14 managerial and related to source code metrics. They conclude that it is not because a model

is able to strongly predict within project that it can predict well in a cross project context.

Cai et al. [2005] classified 18 open source and student applications thanks to source code metrics. This classification is made through summation model, product model, Bayesian belief network among others. According to the authors, these models are intuitive and easy to construct. However, their prediction accuracy is not high.

Jureczko and Madeyski [2010] applied clustering and neural networks approaches on 38 open source, industrial, and students projects. They used source code metrics. The authors obtain a classification of the projects in two clusters. One contains only proprietary projects with a heavy weight development process, a successfully installation in the customer environment, some manual testing, and a use of databases. The other includes proprietary and open source applications with a use of SVN and Jira or Bugzilla, a medium size international team, an automation of the testing process, and where databases are not used. The authors conclude that the identified clusters are far from covering all projects. If clusters are correctly identified, the within cluster prediction will be better than cross cluster predictions.

Turhan et al. [2013] used naive Bayes classifiers on 41 open source and industrial projects with 19 metrics related to complexity, control flow, and software size. The authors concluded that by adding cross project prediction in their within-project model, the results are enhanced, but lightly. The study also found that collecting data from other projects is adding extra effort.

Ratzinger et al. [2007] used eight industrial projects with linear regression, regression trees, and classifier to predict defects. They used 63 evolution metrics related particularly to the size of the project, the team, the complexity of existing solutions,... According to the authors, multiple aspects such as time constraints, process orientation, team related and bug-fix related features play an important role in defect prediction models. They conclude that size and complexity measures have not a major impact on defect-proneness prediction, but people-related issues are important.

Nagappan et al. [2006] described how they apply statistical methods to predict bugs after delivery to the client. They mined five C++ Microsoft applications, including Internet Explorer 6 and DirectX, then correlated 31 code metrics with post-release project failure data. They used module metrics (number of classes or functions), functions metrics (number of lines, parameters, called functions, calling functions, and cyclomatic complexity,...), and class metrics (number of classes, superclasses, subclasses, and classes coupled). By doing statistical analysis, they found, for each project, a set of metrics related to post-release bugs. But, this set is changing from one project to the other. As project failure has to be anticipated, the ideal is to find a unique set of metrics suitable for any project.

Zimmermann et al. [2009] had for goal to predict class defect from both metrics

from source code and project environment. They extracted data from 12 products: closed-source from Microsoft projects, like DirectX, Windows Core File system manager, SQL Server 2005 and Windows kernel, and, open-source like Apache Tomcat, Eclipse, Firefox. Several versions of each system were used for a total of 28 datasets. On each version, 40 metrics were gathered. Concerning source code metrics, Zimmermann et al. used variables ranging from churn (i.e., added, deleted, and changed lines) to cyclomatic complexity. As project environments metrics, they take for example: the domain of the application, the company developing the project, the kind of user interface, the operating system used, the language used, the number of developers. For each metric, they computed median, maximum and standard deviation at project level. Their empirical study gave the following results: on 622 cross-predictions between project tested, only 3.5% of the couples can predict each other. For instance, some models for open-source software projects (Firefox, Tomcat, Eclipse,...) are strong predictors for some closed-source projects but do not predict the other open-source projects. Some other open-source projects cannot be predicted by any of the models in their study. On the closed-source side, they found models for some projects such as File System that can predict some other closed-source projects. However, they also found models for some projects such as Internet Explorer, Windows Kernel, and DirectX that do not predict other projects. There is no way to know in advance which project's model can predict the other projects.

Piggot and Amrit [2013] used open-source software to predict project success. Within the 38 variables analyzed, numerical metrics such as the number of download, of developers or posts in forum better explain success than time-invariant metrics such as license used or the operating system supported. They succeeded to obtain an accuracy of 40% with an exact classification.

Debari et al. [2008] concluded that success is helped when the project team does not use new technology for a project and when the customer is well involved in the requirement specifications.

For Verner et al. [2007], project success is linked to the proper elicitation of the requirements and the accuracy of project cost estimation deduced from these requirements. Success is also acquired by managing correctly the exits of project developers and by avoiding to add new developers too late (*e.g.,* to meet an aggressive schedule).

Cerpa et al. [2016] concluded that what makes the project success is: good working atmosphere for the developers, good communication inside the team and with the customer, good requirements definition, and good staffing management. The working environment has also an impact on the project success.

Wohlin and Andrews [2005] found key success drivers to identify similar projects. The authors concluded that it is a challenge to identify similar projects to predict success of new projects. The perception of the impact of a variable on the predic-

tion model may change over time: the first projects considered for prediction may have to be re-evaluated to ensure that the model keeps a good precision.

1.1.3 Summary

None of these studies is very encouraging. Some of these studies show that it is possible to create a statistical model from one project and have good results by applying it to another project, but they have no way to know in advance if it is going to work.

Each study has its own model (adapted to its context) to predict the success or the failure of the projects. And, each model relies on its own metrics set, even if some metrics can appear in several models. So, finding a set of metrics that can be applied to any project seems unlikely.

Development environments between companies and open-source are different. One is driven by the money and the time, the other is more independent. Moreover, picking the right set of metrics is a major concern. Many metrics exist around projects and depending on the environment, they could be inadequate. We decided to make our own experiment with the company data and metrics.

1.2 Data mining

1.2.1 Methodology

To find what data would be good candidates for our project health predictor, we interrogated seven project leaders on how they manage projects. This was done through unstructured interviews. Each participant is briefly described in Table 2.2 where names are fictitious. The population for the survey was Project leaders and Quality managers of Worldline: they have a high level view on how the projects they supervise are managed. Project leaders manage the project team and distribute the tasks among its members; Quality managers ensure that the projects follow their process. Quality managers have not a fixed team and can be allocated to any project in Worldline.

We had difficulties to find participants for the interviews, but succeeded in finding people from several Business Units (BU) of the company. A business unit is a group of several project teams working in the same business sector. There are three Business Units in Worldline: Merchants Services, Financial Services, and Mobility and e-Transactional Services.

Considering the small number of participants and the very specific purpose of the interviews (discover information on possible predictors available for the projects), we did not see a need for a formal analysis of the interviews result.

There is no standardized form to log project information through the company. However, we found in one business unit some standardized Excel files logging

Table 2.2: Description of the Interviewees

Name	Job position	Experience (years)	Team size
Erin	Quality Manager	25	N/A
Frank	Project Leader	17	20
Grace	Project Leader	6	3
Heidi	Project Leader	10	5
Mallory	Quality Manager	9	N/A
Oscar	Project Leader	19	75
Pat	Quality Manager	23	N/A

business information that we analyzed (see the metrics list below). These projects follow the same development process. Monthly, project leaders fill Excel sheets containing information on their project, bugs encountered, budget already spent, and budget remaining for the rest of the project. Some of the sheets are not filled correctly, because no automatic validity check is made. For instance, the budget or the number of bugs reported can be erroneous. The project can still be in a build phase, *e.g.,* not delivered to the client. In this case, the number of post release bugs is not set in the sheet, then we decided to discard the Excel file of this project. It was the case for around half of the projects.

We used an Excel parser to convert Excel data into a Moose[1] model. Moose is a Pharo[2] based extensive platform for software and data analysis. This platform, based on meta-modeling principles, offers multiple services ranging from importing and parsing data, to modeling, measuring, querying, mining, and building interactive and visual analysis tools.

The usefulness of such a model is to standardize the data and make it accessible in one place. Once the data are modelised thanks to Moose, we conduct the statistical analysis with R[3].

R is an open-source platform to apply statistics tools and algorithms on a large set of data. It is widely used by statisticians to perform data analysis.

In the Excel sheets, we have 12 project metrics available for each month:

Bugs recorded and categorized in terms of:

Seriousness: critical, major, minor. A critical bug impedes the usage of a functionality of the application whereas a minor bug can be a misplaced button in the user interface. A major bug is in between.

Testing phases: qualification, acceptance, or production steps. These metrics correspond to the number of bugs found in each phase. The qualifi-

[1] Bhatti et al. [2012]
[2] Black et al. [2009]
[3] R Core Team [2013]

cation testing phase is done by the company employees while the other steps are led by the client. The production testing is realized in real conditions with the end-users.

Budget: We found two metrics related to it:

Predicted project budget: It is the budget (in *man-days*) set at the beginning of the project. For each month an estimate of the budget is done depending on the progress of the project and the number of developers working on it during the month.

Realized project budget: It is the budget effectively spent during the month on the project. It is known only at the end of the month, whereas the predicted project budget is determined at the beginning of the month.

Slippage: Two metrics to characterize it:

Delta between predicted and realized budget: (*man-days* of slippage). It measures the number of *man-days* that the project deviates in the month.

Whether there is slippage or no slippage: There is slippage when the project has at least one month in slippage.

Number of months in slippage: number of months in the project where there is slippage.

If the value is positive, the project has lost days during this month, else it has "saved" some days.

Project name length: It is the number of characters in the project name. It is intended as a "placebo" metric. We compare all results to this metric to see if they can give a better result.

From this data, we computed the following metric:

Delta between the number of qualification and acceptance bugs: For the experiment, we compute this metric as a difference between the number of qualification bugs (company employees testing) and the number of acceptance bugs (client testing). As the qualification and acceptance tests aren't made by the people of the same company, we expect a delta between both values. If the delta is positive, there are more qualification bugs, which is good because it is the project team and not the client who found the bugs. If the delta is negative, more acceptance bugs are found than qualification ones. This will be damaging for the project because of the decrease of the client confidence.

We also added the number of intermediate releases in a project. Projects are split in releases. They represent a milestone in the development of the application. We expect that these milestones may work as "small projects", making the whole project easier to manage and anticipate potential slippage.

As project leaders consider slippage as the most important metric to assess the health of their projects, we used the three metrics related to it. We decided to compare slippage metrics to metrics related to bug number and budget data.

1.2.2 Data Filtering

We used data from 44 projects from 2012 to 2015 to analyze their correlation with slippage. Unfortunately, earlier data were not available. Indeed, normalization of project monitoring has been adopted recently in the department of Worldline where this data were extracted.

Each month, a file is created for each project. Due to the various durations of the projects 1 076 files are available. Each file recalls the data of the previous months since the beginning of the year. So, the December files are enough to get all data for a year. Moreover, data are not complete for more than half of the projects. Consequently, we mined 91 files corresponding to 19 projects.

By the value of their metrics, several projects can have a great influence on the sample. These outliers have to be identified then removed. Statistical methods advise to take out these kinds of extreme values to have a better sample to analyze. We carried a Principal Component Analysis (PCA) to identify these outliers. The PCA algorithm extracts from the initial variables, principal components. Principal components are linearly uncorrelated variables and represents concisely the initial set of variables. The benefit of this analysis is to identify groups of variables and which one are related to the slippage metrics.

Figure 2.1, shows the impact of each individual (project) of the sample on the two first components, *i.e.,* it is the linear projection of the individual on the two components. The first principal component is put on the abscissa and the second on the ordinate. Each point or cross is the projection of the metrics of a project on the two first principal components. In this way, we can determine the outliers. As shown, the two projects represented by crosses are far from the other projects (at the left of the figure). They are the extreme values. We ignored these projects in our study.

1.2.3 Correlation Matrix Results

To have a first intuition, we correlate all metrics described in section 1.2 whose principal characteristics are detailed in Table 2.3.

Figure 2.2 is a correlation matrix of all metrics we analyzed. The crossing of one column and one row shows the correlation value between both variables. It highlights whether there is a linear dependency between 2 variables. If the value is close to 1 or -1, the variables are correlated. In this case, it is likely that the evolution of one will impact the other. The variables are independent if the values are close to 0. The value is negative if variables are correlated but evolve in opposite directions, this kind of correlation is represented by a minus in the cell. As the sample of project is small, we use the Spearman correlation. On the matrix, a lighter colored cell means that the correlation is closer to 0 (no correlation), a darker

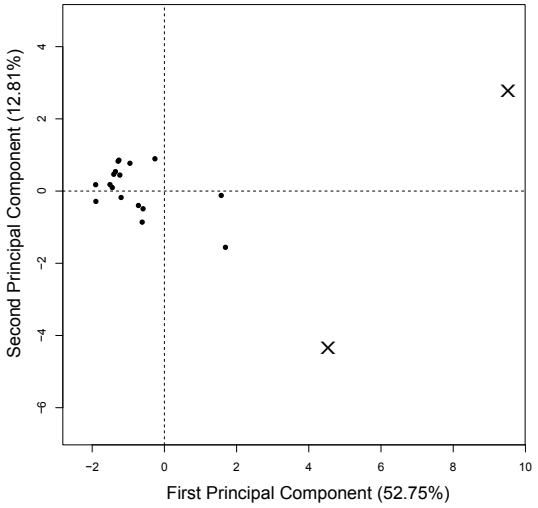


Figure 2.1: Projection of the project metrics on the first and the second principal components

Table 2.3: Statistical summary of the metrics used for the analysis

Metric	Min	Mean	Median	Max
# Qualification bugs	0	20.3	4.0	81
# Acceptance bugs	0	18.3	1	193
# Production bugs	0	8.1	0	120
# Critical bugs	0	12.4	3	101
# Major bugs	0	17.5	11	92
# Minor bugs	0	16.9	7	81
# Total bugs	0	46	25	274
Δ Qualif. & accept.	-112	1.9	0	55
Predicted project budget	31	881	411	4700
Realized project budget	63	947	432	5210
# Months in slippage	0	38.9	34	80
# Man-days of slippage	-60	65.8	32	510
# Intermediate releases	6	18.7	18	31
Project name length	6	18.9	19	32

colored one is closer to 1 or −1 (correlation). The correlation values are detailed in Figure A.1 of Appendix A.

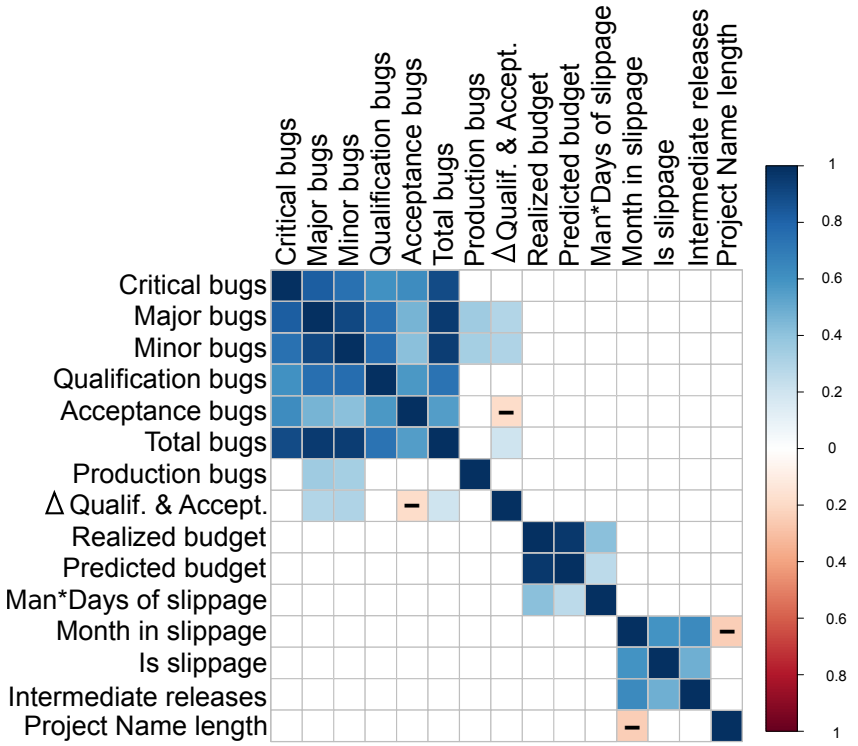


Figure 2.2: Correlation matrix between each metric of the sample

This matrix shows three blocks of correlated variables. First, we can infer a strong correlation between all kind of bugs (the darker square at the top left of the matrix) except the number of bugs in production. These bugs are not strongly correlated to the others.

The metric representing the delta between the qualification and acceptance bugs is lightly correlated to the other ones. The correlation is not strong enough to be considered (between -0.19 and 0.30).

Second, we can also see, in the middle block, correlations between the budget variables: the realized budget and the predicted budget. The number of *man-days* between the initial and final budget is also not correlated to the slippage ones but correlated to the budget metrics. This last correlation might be due to the fact that the bigger a project is, the more difficult it is to predict the budget. A long project is more likely subject to deviations.

Third, it seems also that two slippage metrics (number of month in slippage and if there is slippage or not) are significantly correlated together. The other, the *man-days* of slippage, is not linked to the others. However, the number of

intermediate releases is correlated to these slippage metrics. If more intermediate releases are present in the project, the more slippage there is on the project. It might be the decomposition in group of functionalities that is difficult to determine by the project managers.

Finally, our placebo metric is not correlated to the number of months in slippage (-0.25) according to Hopkins' guidelines [Hopkins, 1997]. The opposite would have been intriguing or even alarming.

However, as shown by the matrix, there is no link between the 3 groups of variables *i.e.,* the bugs, the budget, and the slippage.

In the light of this analysis, we can conclude that there is no link between the slippage and any other studied variable.

1.2.4 PCA Analysis Results

PCA describes more finely the data of the sample than the correlation matrix, in the way that it discovers the internal relationships of the variable. Figure 2.3 displays its result. On the two axes, the first principal component follows the abscissa and the second the ordinate. The abscissa aggregates non-production bug metrics, the ordinate aggregates slippage metrics.

Each arrow represents a variable, the longer the arrow the more this variable is correlated to these two principal components. If the arrow is in the same direction of a principal component, *e.g.,* the first component and the arrow representing the number of critical bugs, both variables are correlated. On the contrary, if the arrow is orthogonal to the component, the variable represented by the arrow is not correlated to the component, *e.g.,* the bug component and the number of slipping months.

For exploratory studies, the normality of the distribution is not required for the PCA. Furthermore, we do not use the PCA results to perform statistical tests [Jolliffe, 1986]. Therefore, we do not need to check that our data are normally distributed.

In our analysis, the first principal component synthesize 43.41% of the variability of the sample and the second 17.51%. It means that almost 60% of the variability of the sample is summarized by these two principal components, *i.e.,* by a linear combination of these two variables, we are able to retrieve 60% of the original data.

The PCA defines four groups of metrics. First, we have the metrics linked to the bugs at the right of Figure 2.3: Critical bugs, Major bugs, Acceptance bugs, Total number of bugs, Qualification bugs, Minor bugs, and Delta between the number of qualification bugs and acceptance bugs (this latest variable is negatively correlated to the others).

Second, Months in slippage, Is slippage, Intermediate releases, and Production

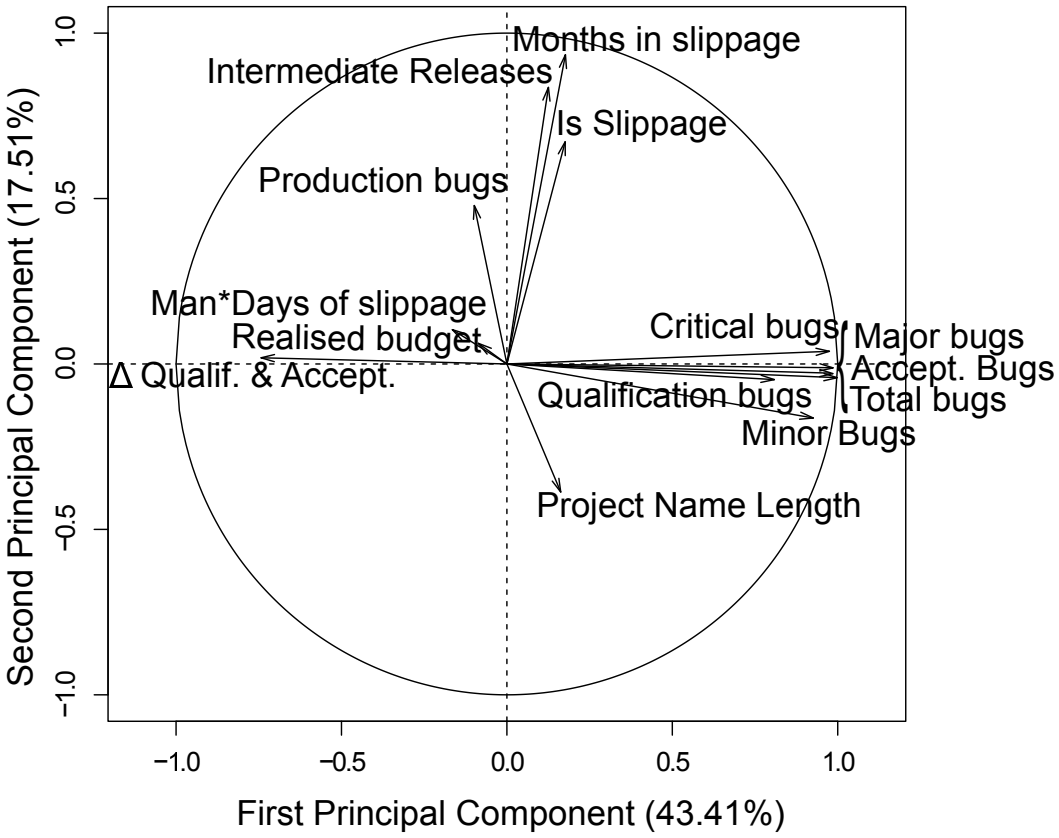


Figure 2.3: Impact of each metric on the sample

bugs variables constitute another group. Despite Production bugs variable is in this group, it is less correlated with the others. Production bugs are bugs revealed by the final customer. A possible explanation, is that if the project is already in slippage, developers are hurried to make the final release, they bypass the best practices, and do not test completely their application. Some bugs are consequently revealed in production. As the PCA shows a finer representation of the data than the correlation matrix, it is possible that a low correlation on the PCA does not appear on the correlation matrix. For example, the relation between the production bugs and the months in slippage are not revealed in the correlation matrix, but is revealed in the PCA.

Third group of metrics, we have the project name length. It is negatively correlated to the slipping metrics. So the shorter the project name is, the more slippage it had. It is alone in its group.

Fourth group of metrics, we can suppose that the two other variables representing the budget (Budget realized, and Delta between predicted and realized budget) make a group of the non-influential variables. The arrows related to these metrics are short on the representation.

As we have the metrics on the intermediate releases, we conducted the same experiment. In the Excel files, only 59 can be used for analysis among the 720. Actually, it is more difficult to get all the metrics for the intermediate releases because the proper filling of the Excel sheets is more complex. After removing outliers, we found the same results than with the complete projects.

To summarize, the correlations we found are quite trivial. Like the papers from the literature survey, we were not able to link the bugs and the budget metrics we used with the slippage metric, considered as an indicator of project health.

1.3 Interviews

As the data analysis and literature review did not show any significant link between project health and project metrics, we conducted four unstructured interviews with experienced project leaders of the company. Frank, Grace, Heidi, and Mallory from the first interviews (see Table 2.2) accepted to answer our questions again. They come from different business units, which allow us to get several points of view on the company. The interviewees gave their feeling on what impacts project health, *i.e.,* success or failure, what are their problems during project development, how they detect them and how they resolve them.

Each interview lasted one hour and was decomposed in two parts. First, we presented briefly the research topic and the context of the study to the interviewee. Second, in an open discussion, we tried to acquire and understand the managers' experience on their project successes and failures. Considering the very small num-

ber of interviews (four), we did not see the need to apply a formal synthesis of their answers. In these interviews, we identified the following root causes of project failure:

Delay at the beginning of the project: if the client decides to begin the project later, the project team is ready to work but is not paid by the client. Consequently, the company spends money and the relationship with the client will deteriorate.

Lack of collaboration between the team and the client: if the team and the client do not know each other well, the collaboration will be difficult and the project is more likely to fail.

Absence of team cohesion: if the members of the project team do not support each other, the cohesion is weaker and the project has significantly more chance to fail.

Misunderstanding the specifications: if the project team does not understand what the client says and fails to transcribe it in his own technical language, the project will progress with difficulty.

Lack of domain knowledge: if the project team does not know the "language" of the client, the project has more chances to fail.

Change of the framework during the development: if the technical tools or the framework, that the project team uses, change, it will cost more to the project.

Lack of experience with the used frameworks: a team without experience on tools or frameworks will be not capable of moving the project faster.

Bypass the qualification tests: if the team does not test its application before delivery to the client, the client will be unhappy because some functionalities will not work and some tension in the project team will appear. As a consequence, the client will find more bugs in the application.

These root causes cannot be mined in project artefacts.

1.4 Conclusion

We conducted a study to check whether software metrics can be related to project failure. The study of literature shows that the metrics extracted from a project cannot be used on another one. The mining of data we have done on company's projects showed no link between project success and data. Moreover, the interviews we conducted shows that the metrics linked to success cannot be found by mining project data.

As all these studies intervene *a posteriori* on projects, it seems random for a new project to know which metric or set of metrics could be used to assess success. Predictive analysis will not work well if it is not possible to know *a priori* which statistical model to use. In this case, there is no utility at all to mine them to predict the health of a project.

Moreover, thanks to the interviews we made, we extracted some topics that are of interest for the project leaders of Worldline: communication, external frameworks, software quality, and tests. To generalize the results of these interviews from four project leaders to all the project members of Worldline, we made a large survey to obtain Worldline's project members insight on project quality.

2 Developers Insight on Project Quality

To get a better point of view on the feelings of the project members about project failure and success root causes, we made a survey for developers, team managers, and architects of Worldline France. Everyday, they are confronting the same problems, loosing time trying to solve them, with or without success. If a generalization can be made, then solving the problem will be interesting for the company in terms of cost savings and wellbeing of the employees.

As health of the project is still an essential factor for Worldline, we focused the survey on the factors that influence the health of the project we discovered previously.

2.1 Survey Description

The participation to the survey is on voluntary basis. To make project members of Worldline answering it, we sent mails to more than 1000 project members and posted on the company social network. We also sent a reminder on both media one month after the first sent and succeed to have 131 project members that anonymously answered to the poll. With 11 questions, the survey took 15-20 minutes to be filled. Questions have been first tried on several beta testers to be sure that they are correctly formulated and understandable, and, conclusions can be draw from the answer. In this survey of 11 questions, seven are focused on the health of the project (the items for each question can be found in the related Figure):

i. On a daily basis, on which criteria do you base yourself to assess the health of your project? (Figure 2.4)

ii. For you, what are the items that are contributing to project success? (Figure 2.5)

iii. Which actions should be taken to improve project health? (Figure 2.6)

iv. What items are contributing to project failure? (Figure 2.7)

v. What are you expecting of a tool that could help you to improve project health? (Figure 2.8)

vi. Which items block you from doing automated tests? (Figure 2.9)

vii. Regarding tests and software quality, which items can help you in the improvement of your project? (Figure 2.10)

The four others questions describe the respondents in the purpose to cross data: company department, responsibilities, years of experience, years of experience inside the company.

The first seven questions are based on a likert scale. For each of these questions, several affirmations are given and the respondents must answer his agreement or not in a scale to define his feeling. The scale specifies several answers to a given affirmation: *e.g.,* strongly disagree, disagree, agree, strongly agree. Moreover, we choose likert scales with an even number of answers. It forces the respondents to have a distinct opinion and to make his mind on one side: agree or disagree.

For each question, the respondents is able to answer "NA" if the question is "Not applicable" or if it does not want to give an answer ("No Answer"). Also, each question can be commented to give an other option or opinion.

2.2 Results

For each question, we focus only on the three or four items with the highest score. Consequently, others conclusions that could be drawn from this survey will not be explained nor studied here.

2.2.1 On a Daily Basis, on Which Criteria do you Base Yourself to Assess the Health of a Project?

It seems that being ahead of time on the schedule is at least an important criteria to measure the health of a project for 93% of the respondents (see Figure 2.4). The relation with the client comes in second place with 86% that think it is an important criterion. In third position comes the software quality with 84%. Fourth is the number of bugs detected in customer acceptance with 81%. The bugs detected in customer acceptance are bugs that should be avoided by the developers. They give a low opinion of the project team to the client and can damage the relation between them.

A high number of bugs in customer acceptance shows a lack of quality in the software. The interviews of our previous study concluded that a root cause of the failure of the project is the high number of bugs listed by the clients. This survey confirms this first intuition. While we are not able to manage the relation with the client in this study, reducing the number of bugs seems to be obtainable by helping the developers to check their application before releasing to the client.

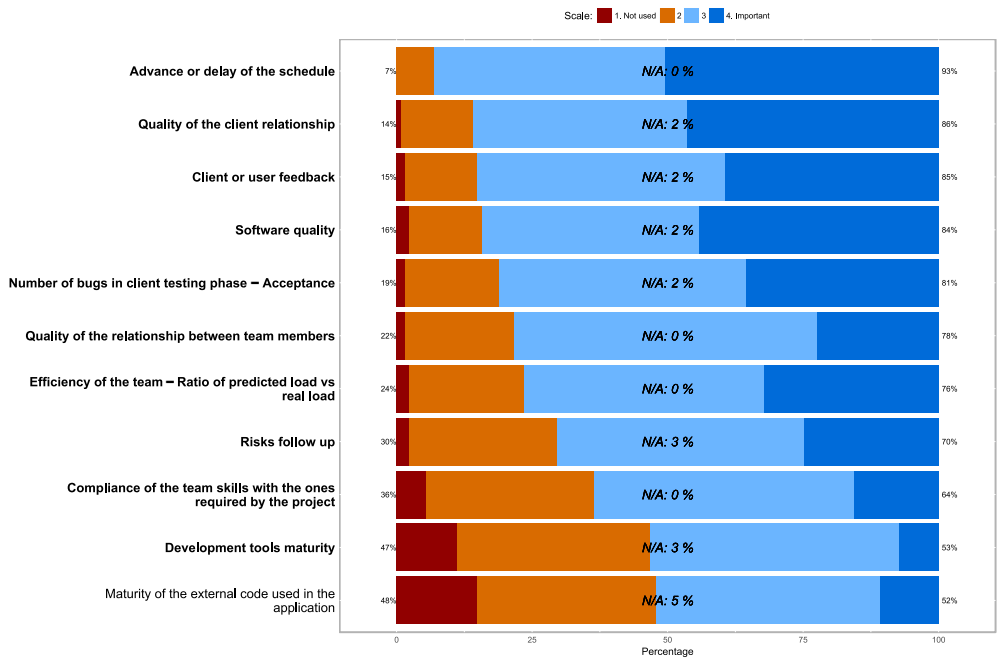


Figure 2.4: Question 1: On a daily basis, on which criteria do you base yourself to assess the health of your project?

2.2.2 For you, what are the Items Contributing to Project Success?

We want to understand what makes a project succeed. As first factor, understanding of the client specifications is essential for the respondents, 98% thinks that this influences project success (see Figure 2.5). As second factor, communication is important for the respondents, the communication in the project team and its cohesion (resp. 96% and 95%), and the expertise of the project leader (93%) have an influence on project success. As third factor comes tests before release and source quality (with resp. 87% and 81%). It is important to note that almost half of the respondents thinks that running tests before release has a *high* influence (whereas other figures are only about influence). However, 76% of the respondents think that having automated tests has an influence of the project success. Like the previous question, communication between the project entities and the definition of the specifications with the client is out of reach of our study. However, we can improve project quality and make sure that the tests are launched before releasing to the client.

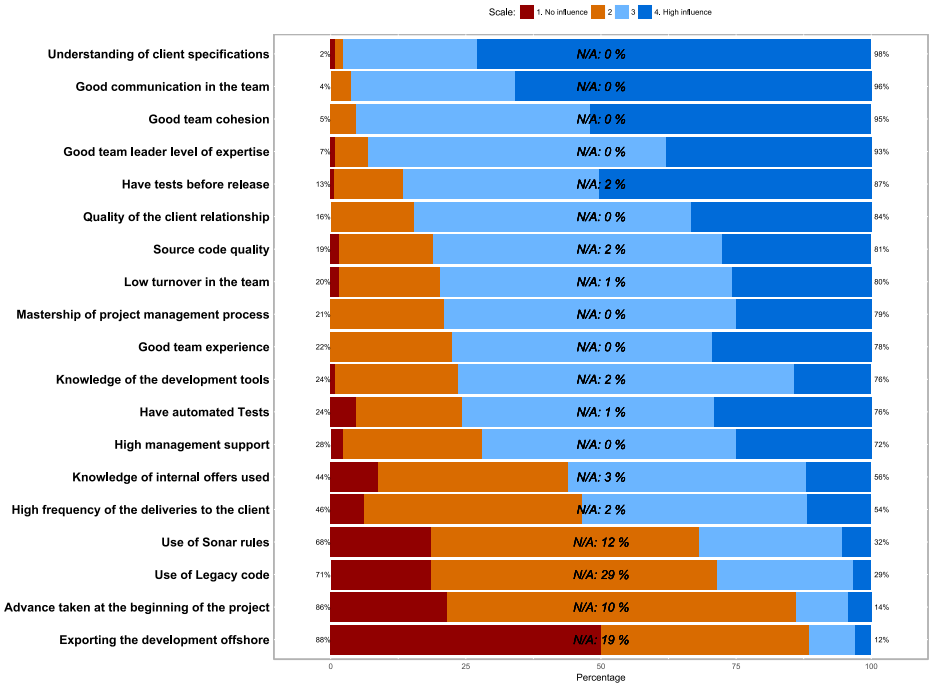


Figure 2.5: Question 2: For you, what items are contributing to project success?

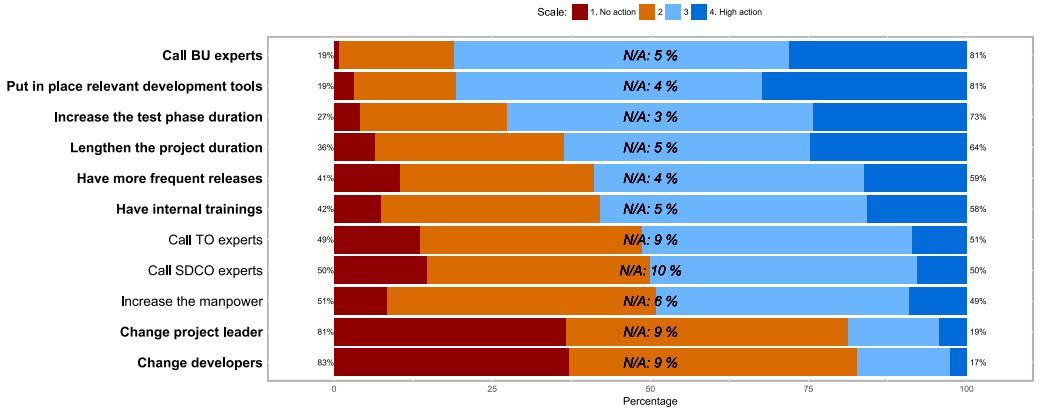


Figure 2.6: Question 3: Which actions should be taken to improve project health?

2.2.3 Which Actions Should be Taken to Improve Project Health?

The first action that should be taken to improve project health is to call the business unit (81%) (see Figure 2.6). Each business unit has experts, by opposition to transversal experts (TO and SDCO experts). BU experts are specialists of the business domain and can solve the issues most frequently encountered in the business unit. TO experts manage the production infrastructure and can help solving related issues. SDCO experts provide support in software and methodology. They are experienced enough to know how to solve the most frequent issues they encounter in the Business Unit. They can indeed help to straighten up the situation of the project. Putting in place development tools is also important for improving the health of the project (81%). Developers want tools that ease the development of the application. Good development tools avoid to the developers to write unnecessary code or help them finding a solution to their problem faster.

In third and fourth position came the increase of the project duration. Firstly is the increase of the tests phase duration (73%) and secondly the overall project duration (64%). During the testing phase some critical bugs can be discovered. Moreover, when project is in slippage, the testing phase tends to be shortened or bypassed. From the previous study (see Section 1.3), we identified the delay at the beginning of a project as a root cause of project failure. If there is such a delay, the project duration is shortened and we can suppose that project members will not have enough time to develop the application. Thus, failure is more likely.

2.2.4 What are the Items that are Contributing to Project Failure?

We took exactly the same criteria that the question: “For you, what are the Items Contributing to Project Success?”. But, we inverted them to focus on project failure instead of project success. A criterion can contribute to the success of the project, but its absence can have no impact on its failure. That is why, we added this question. 90% of the respondents thinks that lack of communication in the team has a prominent place in the failure of the project (see Figure 2.7). The lack of tests before a release has also an influence on the project failure (88% of the respondents).

2.2.5 In Order to Improve Project Health, what is the Purpose of the Tool that Could Help you?

The goal is to know what tool the developers would like to have to improve the health of the project they are working on. Following the project quality is the most desired feature (86%) (see Figure 2.8). Such a tool implies to provide a dashboard to the project leaders. This way, they can monitor main metrics on their projects. A transversal team of Worldline supplies already the project teams with tools allowing

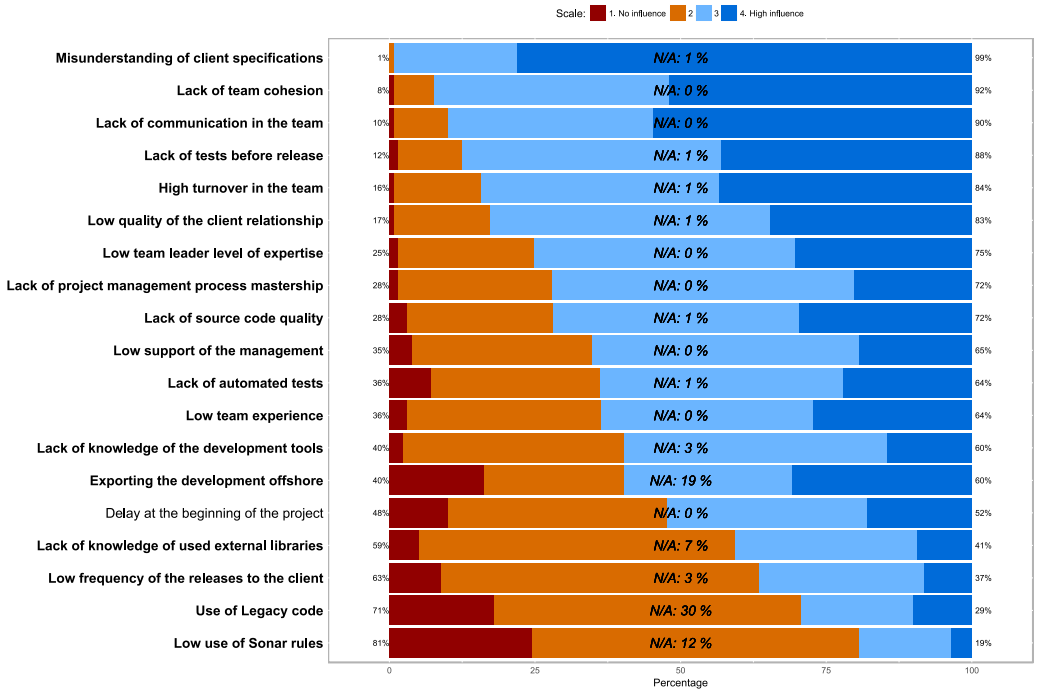


Figure 2.7: Question 4: What are the items that are contributing to project failure?

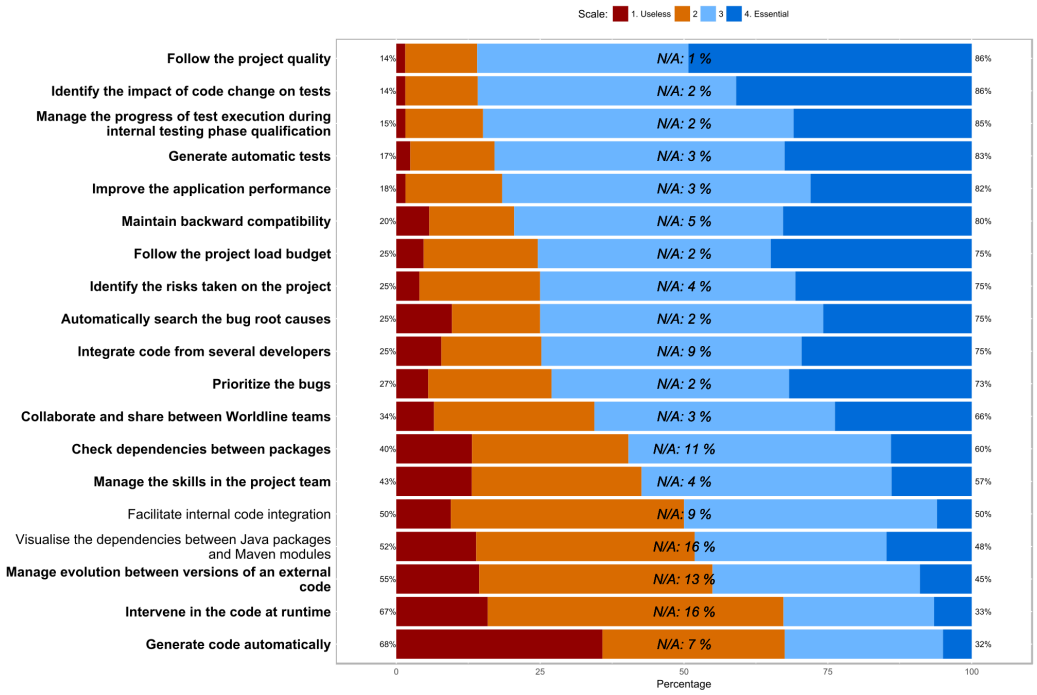


Figure 2.8: Question 5: In order to improve project health, could a tool help you to:

to monitor the project quality. Maybe, the tools are not adapted for their usage that requires specific plugins, or project leaders do not want to use them.

Testing is the second item that can improve the project health. For the respondents, the useful problems that a tool can help to resolve are: Identify the impact of code changes on tests (86%), manage the progress of test execution during internal testing phase (85%), and generate automatic tests (83%). Help testing the projects is an item desired by the developers. We also concluded with our previous interviews (see Section 1.3) that spending too few time on the tests had a negative impact on the project success.

2.2.6 Which Items Block you from Doing Automated Tests?

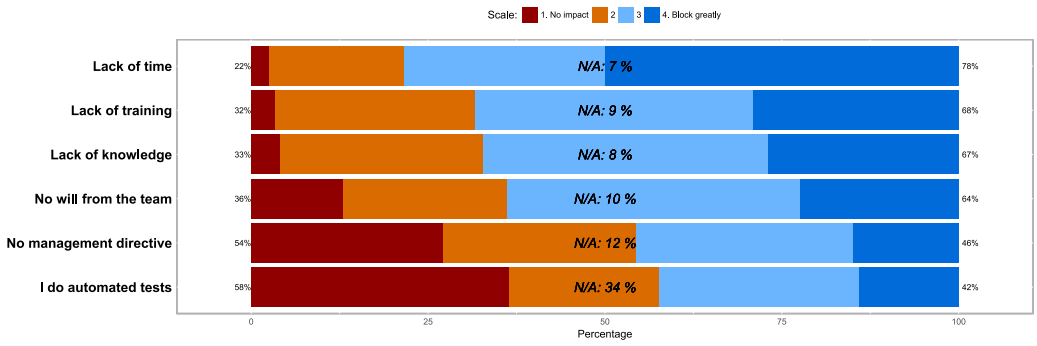


Figure 2.9: Question 8: Which items block you from doing automated tests?

Lack of time, lack of training, and lack of knowledge are the main blockers with resp. 78%, 68%, and 67% of respondents who thinks that these items block them from doing automated tests (see Figure 2.9). Indeed, like shown in previous questions, the lack of time is frequent in projects. Initiatives have been set up to train people and add knowledge on this topic.

2.2.7 Regarding Tests and Software Quality, which Items can Help you in the Improvement of your Project?

Automation of tests execution and detection of tests after a code change are the 2 items that are found useful with resp. 91% and 87% (see Figure 2.10). These items are important for the developers. However, the automation of test execution is complex to set up, due to the high number of frameworks used for testing in Worldline, to the complexity of the applications and to the use of legacy code. The training of the developers is in progress but still not generalized. Knowledge is also not shared easily between the projects teams.

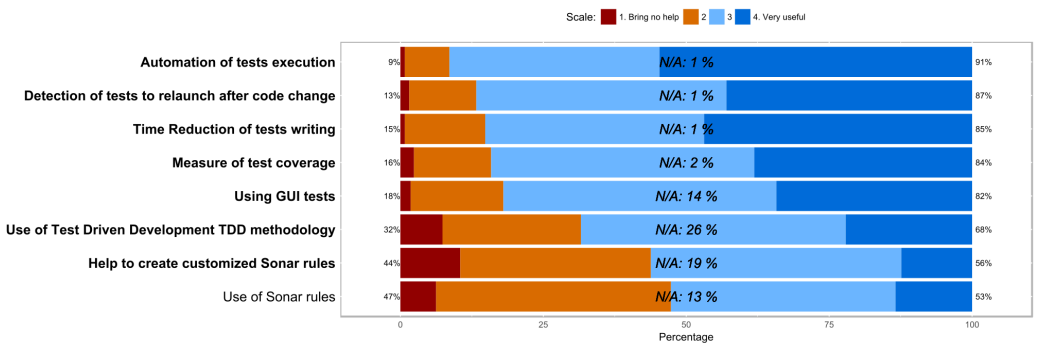


Figure 2.10: Question 9: Regarding tests and software quality, which items can help you in the improvement of your project?

From previous questions, developers thinks that the automation of the execution of the tests is important. Moreover, as they lack time to test their application, they often bypass this step. The second item, detecting the tests after a code change, could allow them to spend less time to test their application.

2.3 Conclusion

To conclude, we made a survey on 131 projects members of Worldline (developers, architects, project leaders...). This survey confirms the results of our previous experiment at a higher scale: The understanding of the specifications by the client is primordial, as is the communication between the project team and the client and inside the team. The testing of an application is also an important topic that can make the success of the application. However, testing requires time. But, interviewees said that when the schedule is shortened on the project, testing is the first part of the project that is bypassed, as it is the last part done on a project. From the last question of the survey, it seems that detecting automatic tests after a code change can help project members to improve their project. So, they will be able to shorten the test execution duration and, consequently, improve the quality of their application before delivering to the client.

Proposing such a tool to the developers can help them to ensure the health of their project. We hope that this would motivate them to make more automatic tests and test more often.

But at first, test selection approaches have to be designed for Worldline environment: first by respecting the constraints of the language and frameworks they use, second, by knowing the testing behavior of Worldline developers to ensure a good adaptation of the tool to their practices.

CHAPTER 3

State of the Art

Contents

To design a specific test selection approach for Worldline's projects environment, literature needs to be studied. In a first part, we analyzed test selection approaches, then we compared some tools available for test selection in the IDE of the developers, and finally looked into studies on the monitoring of developers on software testing.

1 Test Selection Approaches

Test case selection techniques aim to reduce the number of test cases to execute after modifying code in order to avoid to launch all the test cases and spend less time to test their application. Test cases are selected if they are relevant to the changed parts of the system under tests [Yoo and Harman, 2012]. Thus, the selection is not only temporary (*i.e.,* specific to the current version of the program) but also focused on the identification of the modified parts of the program. More formally, following Rothermel and Harrold [1993], the selection problem is defined as follows:

> *Definition Test case selection problem*
> *Given*: The program, P, the modified version of P, P' and some test cases, T.
> *Problem*: Find a subset of T , T', with which to test P'.

Literature (*e.g.,* Engström et al. [2008, 2010], Ernst [2003]) recognizes several types of approaches for test case selection. Kazmi et al. [2017] classify them in five main techniques:

Mining and Learning According to Kazmi et al. [2017], it is the most frequently studied approach. Genetic Algorithms are used to select the tests. They use

learning algorithms which can lead to optimal solutions. But, a part of randomization has to be added in the approach, thus, they do not give exact results. These approaches do not scale well and thus cannot be used in companies: Kazmi et al. [2017] advances that *"Mining and learning techniques have several drawbacks like small size datasets and high computational cost that made it unattractive to industry players."*

Model Based Testing In this approach, UML diagrams are used as ground to select the tests. This kind of approach works only if the source code is synchronized with the model. In Worldline, UML generation is used at the beginning of the project. Nevertheless, keeping the model and the source code in synchronization is challenging. Even if the project has taken the approach to use UML code generation at the beginning of the project, later, only the source code is changed. The model is not updated and cannot be used to retrieve the tests.

Program Slicing These are the pioneer Regression Test Selection (RTS) techniques. They simplify the program such as to omit test cases that do not produce new and different execution traces. This approach lacks exactness and have been only applied to small or medium-sized data sets, which are mostly procedural applications. It is consequently not appropriate for Worldline projects that use object oriented languages.

Control Flow Graph based RTS These techniques are based on flow control of programs, workflow of programs behavior, or functions. These approaches are stable and are adapted to existing tests because they focus on already implemented program flow.

Oracle based RTS These approaches try to predict the effectiveness of the test suites for future uses. To be efficient, these approaches need an history of the launched tests. For Ekelund and Engström [2015], an history of 100 build is the optimum. But, such an history is not available for the Worldline projects. However, we compare in Chapter 6 the Control Flow Graph approaches to the one of Ekelund and Engström [2015]

Consequently, the Control Flow Graph approaches seem the best to the Worldline problem of test selection. We focus our literature study on it.

1.1 Control Flow Graph Approaches

The general idea is the following: a test depends on a piece of source code if, by launching this test, the piece of code is executed. It can be directly executed through method calls. After a piece of code is changed, the test case selection technique should select only the tests depending on this piece. For this purpose, the approach navigates the control flow graph back from this piece to the tests that depend on it. Figure 3.1 illustrates this principle for two methods and two tests. `testMethod1`

depends on `method1` and `method2` (for example `testMethod1` calls `method1` and `method2`), `testMethod2` depends on `method2`.

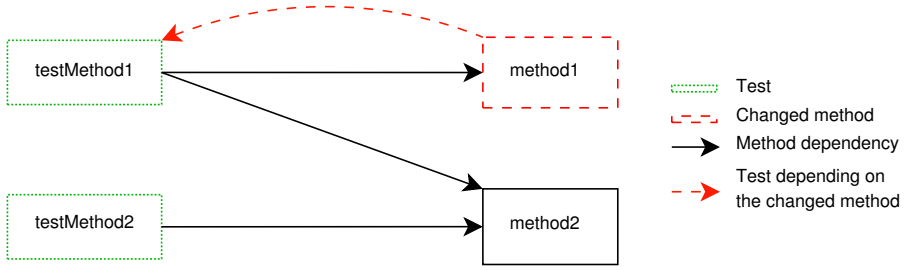


Figure 3.1: Test Selection Simple Case

This is, of course, a simplified example, in real cases, the dependency graph is much larger and deeper, or, some other factors may make it very difficult for a given approach to find out which tests depend on a piece of code.

Control Flow Graph approaches are split into two categories: Dynamic and Static approaches.

The *dynamic approach* consists in executing the tests and recording the code executed during each test. This is the execution trace of a test. A test depends on a piece of code if this piece of code is in its execution trace. For example, by using this approach on Figure 3.1, a dynamic model contains a mapping for `method1` to `testMethod1`, and for `method2` to `testMethod1` and `testMethod2`. If `method1` is modified, the mapping stored in the model is used to select only `testMethod1`.

The *static approach* does not require executing the tests. It relies on computing the dependency graph from the source code or some representation of it (*e.g.,* bytecode for Java). Several dependency graphs can be used [Biswas et al., 2011, Engström et al., 2008, 2010]: Data dependency graph, Control dependency graph, Object relation diagram, etc. For example, by considering arrows in Figure 3.1 as calls between methods, a static approach will create a static model of the source code where `method1` callers are `testMethod1`, and `method2` callers are `testMethod1` and `testMethod2`. If `method1` is modified, thanks to the static model, the `testMethod1` caller is selected. Note that in real projects, the call graph is much deeper and recursion is used until a test is found.

1.2 Dynamic versus Static: Pros and Cons

Ernst [2003] argues that a static approach guarantees generalization of the results for future executions. This approach usually results in a superset of all actual executions as some combination of cases might seem possible when looking at the

code, but actually impossible in real cases. On the other hand, dynamic approach results in a subset of all actual executions as only some examples are actually run among all the possible executions from a static point of view.

Ernst [2003] also argues that the dynamic approach is as fast as the program execution and does not require costly analyses. It must be noted that in our experiments with the Jacoco tool (see Chapter 4), we found a perceptible increase in time of up to one hour (from an initial five hours) needed to run only the tests. A static approach uses an abstract representation of program state that looses information but is more compact and easier to manipulate than a more faithful dynamic model.

Beszedes et al. [2012] found that the dynamic approach is less reliable if the dynamic model is not updated frequently. Some code addition or modification can impact the accuracy of the approach if tests are not run again to recompute their new execution trace. However, updating the execution traces after each change is in complete opposition with test case selection that aims to reduce the test set, and avoids launching all the tests. Therefore, a compromise must be found between an up to date dynamic model (to get good results) and actual test selection. Finally, if the test setup (executed before all tests) or an instruction of a test fails, the execution is stopped and only a part of the code is exercised.

Ekelund and Engström [2015] consider that the creation of a static model can sometimes be a drawback. In case of large systems, representing the whole source code can be costly. Ekelund and Engström had to give up static approaches because of static model creation time. This statement should be nuanced with our experiments, where running static approaches is 12 times faster than running the tests with Jacoco, *i.e.,* about 30 minutes versus six hours.

After an analysis of 36 studies on test case selection, Engström et al. [2010] conclude that the empirical evidence for differences between the techniques is not very strong, and sometimes contradictory. As summary, there are no bases to select a technique as superior to the other. Techniques have to be tailored to specific situations.

But, Legunsen et al. [2016] evaluated static test case selection against dynamic one. Static RTS could be more beneficial than dynamic test case selection for systems with long-running tests, non-determinism, or real-time constraints. It shows promising results when used at the right granularity.

1.3 Evaluation Criteria

We need to choose between the existing Control Flow Graph approaches to find one adapted to Worldline. Some criteria are used to compare them:

Granularity. Different kinds of granularity can be considered [Engström et al., 2010] from individual instructions (*e.g.,* Rothermel and Harrold [1993]) to modules (*e.g.,* White and Leung [1992]) or external components (*e.g.,* Will-

mor and Embury [2005]) passing through functions/methods (*e.g.,* Elbaum et al. [2003], Zheng et al. [2007]) and classes (*e.g.,* Hsia et al. [1997], White et al. [2005]). Using a smaller granularity gives better precision but is more costly [Engström et al., 2010].

Evaluation. The evaluation of the approach can concern open-source or closed source projects. Both kind of projects can have their own particularities that change the result for the comparison of the approaches.

Approach. The approach used to select the tests, whether it is a static or a dynamic one.

Object Oriented Programing (OOP) Friendly. A large part of the programs developed in Worldline uses Java and C++ which are Object Oriented Languages. The appropriated test selection approach must take in consideration the specificities of the language. Some issues are specific to OOP while some others are not present (*e.g.,* the pointers in C).

1.4 Test Selection Approach

Some software representations can be used as basis for static analysis: First, Badri et al. [2005] present a static approach for predictive change impact analysis. It is based on control call graphs but it is more precise than the standard approaches based on call graphs. No execution of the program is required and this kind of approach may be used for test selection. They do not consider the specificities of object-oriented systems in their study.

Jász et al. [2008] compare several control flow approaches they defined (SEA, Static Execute After, and SEB, Static Execute Before) to a traditional one: System Dependence Graph (SDG). These graphs represent dynamic dependencies between the call of methods. Whereas SDG uses both control and data dependencies, SEA and SEB use only control flow. The authors carried out a study on four open source projects (Valgrind, gcc, gdb and Mozilla), where source code is written mainly with procedural languages (C++ or C). Their experiments conclude that SEA and SEB precision is lower but computation is more efficient than SDG. Jász et al. approaches could be used to represent the source code at a lower granularity level.

Ekstazi is a lightweight RTS technique [Gligoric et al., 2015]. This tool is based on a dynamic approach and tracks dynamic dependencies of tests on files. Ekstazi is able to select regression tests from a changed method. It is now integrated in some popular open-source projects like Apache Camel. Evaluated on 32 open-source projects, Ekstazi allows to reduce globally the testing time of 32%, and reduces it of 54% for longer running test suites. Moreover, an experiment on 20 revisions per projects on around 20 Java projects is performed at class and method granularity. The ratio of selected tests is lower at method granularity (8%) than at class granularity (11%). However, testing time is a lot more reduced in case

of class granularity. Furthermore, the approach requires to run all the tests which could lead to a lost of time.

Hurdugaci and Zaidman [2012] developed TestNForce, a tool integrated in Microsoft Visual Studio that shows to the user the unit tests to relaunch after a change. The goal is to help developers better co-evolve test and production source code. The tool works on C# applications thanks to an instrumentation of the binaries after a compilation of the source code. Each time a test is launched, its coverage is recorded. But, building such a coverage takes a long time: 28 minutes are needed for a project containing 344 unit tests. This tool has been tested in an experiment with 8 students of the Delft University of Technology. Participants were asked to try the tool on a given project and perform a set of defined programming tasks. Students found the tool useful for their test maintenance activities with a score of 4.5 on a 5-point likert scale. They also found that the normal usage of TestNForce is not an hindrance in the development process. But, a feature has been implemented to prevent commit to the repository if the tests related to the changes are not executed or updated. According to the authors, the presence of this feature has decreased the average score hindrance of 2.125 on 5.

Soetens et al. [2013] propose a static approach at method granularity based on the FAMIX meta-model. Their approach relies on real change sets gathered in commits. For each of these change sets, their static approach is compared to a dynamic one used as reference. The dependency graph they use only contains links between methods. Two open-source applications (PMD and Cruisecontrol) are used as input data for their experiment. 1% of the test cases is selected for both applications. They obtain respectively for each application a recall of 77% and 58% and a precision of 84% and 83% (see Section 2.4 for the definitions).

These approaches are summarized in Table 3.1.

Table 3.1: Approaches Criteria Matrix

	Granularity	**Evaluation**	**Approach**	**OOP Friendly**
Badri et al. [2005]	Instruction	Open-source	Static	No
Jász et al. [2008]	Instruction	Open-source	Static	No
Gligoric et al. [2015]	Instruction	Open-source	Dynamic	Yes
Hurdugaci and Zaidman [2012]	Method	Open-source	Dynamic	Yes
Soetens et al. [2013]	Method	Open-source	Static	Yes

2 Tooling for Test Selection

Some test selection tools are already available for the developers but are not yet propagated inside the company, *e.g.,* Infinitest or Clover. We identified some tools that can interest the company and classified them with criteria to compare them.

2.1 Evaluation Criteria

To evaluate the tools selecting tests, we defined several criteria. The goal is to identify in a second part which tool would be more appropriate to the Worldline environment. The criteria we used are:

Approach. It defines the approach of the test selection. The goal being to compare static and dynamic approach, this criteria is important to know which approaches have to be compared together.

IDE Integration. If the test selection tool can be integrated in the IDE of the developers, the selected tests can be launched automatically. If this feature is proposed, the tool could be directly suggested to the developers.

Open-Source. For our study, we will need to understand the internals of the tool and possibly modify it. If some issues or some blockers are revealed, an open-source tool eases these modifications.

Licensing. The cost of the plugin should be taken in account for the evaluation of the tool. A tool with a charged license cannot be envisaged for Worldline.

2.2 Tooling

The following tools are described just after and are positioned in the criteria matrix in Table 3.2.

Jacoco

Jacoco[1] aims to compute the test coverage of an application [Lingampally et al., 2007]. For this purpose, the Java Virtual Machine (JVM) is instrumented by adding an agent to add behavior to the source code and to record method dependencies during the tests execution. No recompilation nor modification of the source code is needed. However, a synthesis of the results is needed after the execution. It can have an impact on the execution time. For instance, for the studied projects in Chapter 4, about one hour is needed for this synthesis for each project. The test coverage tooling is integrated in the IDE, but a test selection tooling based on this approach does not yet exists.

However, the data provided by Jacoco is not directly usable for test selection because information concerning the executed tests are mixed up. Quite a bit of modifications had to be done to transform it in an actual test selection tool.

[1] http://eclemma.org/jacoco/

Clover

Clover[2] is a Java Code Coverage Analysis application. The application was made open source in 2017, source code was not available during my thesis. This tool uses a source code instrumentation technique like Jacoco then records precisely what is executed when tests are run. The detailed test coverage reports help developers easily identify areas where the testing is weak, enabling them to write optimal tests.

The user triggers the test selection when he pushes a button. All the tests that cover the last changes are then selected. However, tests should be first launched to know their coverage of the source code to be selected for automatic launching. As any dynamic approach, the tests selected by the tool have to be launched and their coverage updated in the model to stay up to date.

Infinitest

Infinitest[3] is a test selection plugin which analyses the bytecode using a static approach. Integrated in a Java IDE (Eclipse or Intellij), it intends to do test case selection, but, through the experimentation we made on complex projects, we saw that the algorithm is slow. This algorithm relies on an internal representation of the application source code. This representation must be computed each time the IDE starts. A project of 300K lines of code can take several hours to load. So, it is not satisfactory considering Worldline projects size.

Infinitest works at class granularity level. For a given class, it gathers all references to other classes (*i.e.,* the classes of the methods invoked, the types used, the annotation types...) and thus provides a graph of class dependencies. It also selects tests at the class level, that is to say not individual test methods, but their classes.

Moose

Moose[4] [Ducasse et al., 2000] is a tool allowing test selection in a static way. It analyses the source code. Moose relies on the FAMIX meta-model [Ducasse et al., 2011] and proposes to represent source code entities in a model. This model gathers entities such as packages, classes, methods, and the links between them (invocations, references, inheritances, and accesses). A method dependency graph, linking a changed method to the tests, is thus available.

Moose is a tool dedicated to pure static analysis, and so does not need any compilation of the source code. However, the source code has to be parsed to create the model. This parsing can take up to several minutes for large applications. In our

[2] https://confluence.atlassian.com/clover/
[3] http://infinitest.github.io/
[4] http://www.moosetechnology.org/

experiment done in Chapter 4, the result overhead was smaller than for the other approaches.

Table 3.2: Tools Criteria Matrix

	Strategy	IDE Integration	Open-Source	Licensing
Clover	Dynamic	Yes	No[5]	Charged
Ekstazi	Dynamic	No	No	Free
Jacoco	Dynamic	No	Yes	Free
Infinitest	Static	Yes	Yes	Free
Moose	Static	No	Yes	Free

3 Testing Habits of Developers

We need to enhance our comprehension of our industrial environment and to ensure that the practices of Worldline developers are known. Observational studies monitor and conduct interviews with the developers to acquire their behavior inside their IDE.

3.1 Evaluation Criteria

We defined criteria to classify the papers of the literature:

Participants. Studies have mainly three types of participants: students, employees, or open-source developers. Literature showed that conclusions might be different in those environments Zimmermann et al. [2009].

Project Size. The size of the application on which the studies are based can lead to different results too.

Recording. Another criteria is the detail level of the recording. Only the test operations can be logged, or other interactions with IDE can be added.

Interviews. The studies can report some interviews on the participants of the experiment. It give more weight to the results and confirm the quantitative results the recording gave.

 # Participants. The number of participants to the study. The more participants the study includes, the more the results can be generalized.

Language. Depending of the programming language the developers use, their behavior about testing can be different. Some languages are more prone to be tested than others.

[5] Open sourced in April 2017

3.2 Studies

Table 3.3 summarizes the studies on developer testing behavior we identified in the literature.

Table 3.3: Criteria Matrix for Study of Developer Test Behavior

	Participants	Project Size	Recording Detail level	Interviews	# Participants	Language
Munir et al. [2014]	Industrials	Small	N/A	Yes	31	Java
Kasurinen et al. [2010]	Industrials	Various	N/A	Yes	55	N/A
Runeson [2006]	Industrials	Various	N/A	Yes	24	N/A
Pham et al. [2014]	Students	Small	N/A	Yes	97	Java
Pinto et al. [2012]	Open-source	Various	N/A	No	N/A	Java
Zaidman et al. [2011]	Open-source + Indus.	Various	N/A	Yes	6	Java
Amann et al. [2016]	Industrials	Large	Interactions	No	84	C#
Gligoric et al. [2014]	Stud. + Indus.	Mainly small	Tests + Interactions	No	14	Java
Beller et al. [2015]	Open-source + Stud. + Indus.	Various	Tests + Interactions	No	416	Java

Munir et al. [2014] made a controlled experiment to compare the impact of Test-Driven Development (TDD) on internal code quality, external code quality and productivity against Test-Last-Development (TLD). Thirty-one developers from industry, with at least one year of experience, were asked to develop code according to user stories. Fifteen were assigned to TLD tasks and 16 to TDD. Results of the experiment clearly indicate that TLD is easy to use (100% answers) and easier than TDD (86% of the answers). TLD will also be selected as the first choice development method for 38% of the participants. The authors conclude that TDD adoption requires not only a strict discipline to actually write the test first but also an adequate and sufficient training in improving developers skill set in testing.

Kasurinen et al. [2010] interviewed 55 industrials from 31 companies and studied 12 software systems in development. Their survey revealed that organizations use automated testing only in 26% of their test cases. Based on their study of the literature, it is considerably less than the authors expected. The results indicate that test automation is in demand in software organizations. The lack of a global strategy for applying automation was also evident in many organizations they interviewed. These observations also indicate communication gaps between stakeholders of the overall testing strategy, especially developers and testers.

Runeson [2006] conducted a survey on companies employees. The goal was twofold. First the survey aims to clarify through interviews what is a unit test. Second, the survey evaluates the strengths and weaknesses of unit tests. Seventeen industrials participated to a focus group and 15 filled in a questionnaire. In total, 24 unique industrials where interviewed from several companies. A clear and shared definition of unit testing enables to better determine the role and responsibilities of the involved stakeholders. Interviewees report the complexity to test GUI modules, to identify the unit to test and to maintain tests.

Pham et al. [2014] conducted a study with 97 computer science students and made interviews to explore their attitudes regarding testing in a collaborative software project. Students tend to push test automation to the end of the project and consequently avoid to have a test suite during the development. The authors explain that it is mainly because of lack of time that they do not become productive with testing. We felt the same behavior with the employees of the company. Due to the tight schedules, testing is often left out. However, a majority of the interviewees think that the tests are essential in the development of their applications.

Pinto et al. [2012] proposed a technique to study test suite evolution. They developed a tool named TestEvol that mines source code repositories to study these evolutions. An empirical study was done on six Java open source software of the real world: Gson, PMD, JFreeChart, JodaTime, Commons Lang and Commons Math. According to the study, modifying tests to make them pass corresponds to only 23% of the test changes. The authors also conclude that failing tests are deleted not because they are difficult to fix, but because they are obsolete. New tests are added to check bug fixes, test new functionality, and validate changes made to the code.

Zaidman et al. [2011] studied the co-evolution between production code and test code. They conceived a tool to visualize these co-changes to help developers better know their practices and improve them. The authors analyzed three projects: two open-source, Checkstyle and ArgoUML, and one industrial, SIG. They surveyed 2 persons on each project to gather their insights. The authors' case studies do not emphasize an increase in testing activity before major releases, but periods of intense testing in the development's history.

Amann et al. [2016] studied the general usage of the Visual Studio IDE. They tracked the interactions with the tool of 84 professional C# developers in an industrial environment, combining 6 300 hours of work time. They found that unit-testing tools are rarely used. They mention a tool (NCrunch) that automatically runs tests on identified code changes and displays the results. They estimated that 9 developers (11%) used this tool for a total of 21 developer days (2%[6]) whereas testing tools are "used on little more than a fourth of all developer days." NCrunch (only available for VisualStudio) matches the solution we wish to implement in the company: it runs tests in the background to pro-actively give feedback to the developers. However, Amann et al. [2016] describe their test selection mechanism as still experimental and very rudimentary[7]. We have no information to explain its low adoption in the experiment, but foremost, we have no information on its consequences on the testing habits of the developers. Nevertheless, it has to be noticed that Visual Studio 2017 newly integrates a tool that automatically selects the tests

[6]Our statistics from their numbers.

[7]From their web page http://www.ncrunch.net/documentation/concepts_engine-modes, 08/23/2016

to relaunch after source code modifications.

Beller et al. [2015] study the usage of the IDEs by the developers to understand "When, How, and Why Developers Test". They report on a large scale, field study, with 416 software engineers. They monitor the actions of developers in their IDE thanks to their tool named Watchdog. Anyone could take part in this case study, and it seems that most of the projects were open source but there are also students and industrial projects. Their findings are:

- A majority of the developers rarely test in their IDE (note that they could run tests outside of the IDE). The authors raise several reasons: there are often no preexisting tests for the developers to modify, developers are not aware of existing tests, or testing is too time-consuming or difficult to do. This would indicate that testing is not a common practice. This is a preconception that we have about the company.
- Quick tests do not lead to more test executions. Developers selected test cases whatever the duration.
- Some failing tests are fixed later: 50% of the test repairs happen within 10 minutes whereas 75% within 25 minutes. This indicates a good test practice: tests results are considered. On the other hand, 25% of the failing tests take a long time to be corrected.

Gligoric et al. [2014] compares manual and automated test selection. They assessed how developers manually select tests and compare this manual selection to an automatic one. They conclude that there is a need for better automated test selection techniques that integrate well with developer IDEs. For their study, Gligoric et al. use a group of 14 developers composed of five professional and nine students. They asked them to install a plugin in their IDE which records code changes and test executions. It is possible that students are better trained on regression testing techniques that developers of the company. This paper focuses more on RTS than the previous one and their main findings are:

- Test selection is frequently done (59% of the test executions), and most of the time, the ratio of tests selected is less than 20%. It is important to see that developers routinely perform test selection. We wonder if it applies in our context;
- There is a low correlation between the amount of code changed immediately before a test session and the number of manually selected tests in that session. This finding is in opposition with Beller observations.
- Manual selection results in more tests executed than automated selection in 73% of the cases and results in less tests executed in 27% of the cases. This shows that manual RTS miss to run some potential failing tests. This could be improved by an automated tool. We need to see whether this is the case in the company too.

4 Conclusion

Our conclusions from the literature study: first, control flow graph approaches seems to answer well our need for test selection for Worldline's environment. Two types of approaches exists, the dynamic and the static one. Literature do not conclude on the prevalence of one above the other. Our own study have to be done to chose.

Second, tooling for test selection must be adapted to satisfy the Worldline context. Especially, Jacoco, Infinitest and Moose seems to be good candidates to this adaptation.

Third, studies on developers behavior about testing mainly involve open-source systems or students and may not apply to Worldline. However, their methodology is interesting and can be used as basis for comparison, especially the studies of Beller et al. [2015], Gligoric et al. [2014]. We decided to take inspiration from these papers for our study in Chapter 5.

CHAPTER 4

Comparison of Approaches to Select Tests from Changes

Contents

We need first to find the best approach to select tests adapted to Worldline environment. We saw that, in the literature, two main approaches are proposed: static and dynamic. The static approach creates a model of the source code and explores it to find links between changed methods and tests. The dynamic approach records invocations of methods during the execution of test scenarios. Understanding the issues brought by each approach in the Worldline environment is important.

First, we propose a classification of problems that may arise when trying to identify the tests that cover a method. Then, we give concrete examples of these problems and list some possible solutions. We also evaluate other issues such as the impact on the results of the frequency of method modifications or considering groups of methods instead of single ones. The goal is to know whether or not static approach can be used instead of the dynamic one for Worldline projects.

1 Taxonomy of Issues

To have a test case selection tool adapted to the Worldline environment, we had to evaluate existing approaches.

As most of the projects of the company are written in Java, we therefore limited ourselves to this language or at least to the Object-Oriented paradigm.

The projects use diverse tools and frameworks, that can be off-the-shelf (EJB, Hibernate, Spring, Tomcat...) or proprietary (in-house development). Also these

projects typically use a client/server architecture for the web. Finally, the projects often access databases (mainly Oracle and MySQL) through Hibernate.

Part of the applications are generated through some kind of Model Driven Development (MDD) approach (for more information on MDD see Kent [2002]). Biswas et al. [2011] describe some approaches for test case selection in the presence of MDD. We will not work with them, instead we consider source code representations. This is justified because we found that often in industrial projects, the code has been manually modified after a first generation and is no longer synchronized with the model.

Because of these particularities used in the studied project of Worldline, we were confronted with different issues. We are presenting them here, classified in four categories.

1.1 Proposed Classification of Issues

Problems in test case selection approaches arise when there is a break in the dependency graph representing the system. Such breaks may occur for several reasons. In our case, we identified four categories of reasons. Note that this list might not be exhaustive but contains all the issues we encountered in the company. More concretely, we ran our test selection approach on different projects, evaluated the selected tests and tried to understand why tests were missing. From the different cases, we build the following classification. Some categories may be specific to the static or dynamic approaches.

Third-party breaks: The application uses external libraries or frameworks for which the source code is not available. In this case, a static analysis of the code cannot trace dependencies through the third-party code execution.

Multi-program breaks: The application consists of several co-operating programs (*e.g.,* client/server application). In this case, an analysis focused on one single program cannot trace dependencies into the other program.

Dynamic breaks: The application contains code treated as data (*e.g.,* lambda-expressions or reflexive API). Specific instructions allow to execute this code in another location than its definition. In this case, an analysis of the source code cannot yield the dependencies that will occur at execution.

Polymorphism breaks: The application uses polymorphism. In this case, a dependency analysis may reach a class on which nobody else depends because all dependencies point to a superclass of it.

We now describe the issues we were faced with while investigating with some selection tools.

1.2 Third-Party Breaks

Third-party breaks are encountered when external source code is used in the application. Third-party can be frameworks or libraries. This category only **impacts static approaches** because dynamic approaches can still find in their execution traces the application methods called by the third-party.

This issue is actually protean, *i.e.,* can take several shapes. In some cases, we found that it could be bypassed (see below), in other cases, not so easily.

1.2.1 External Source Code

Context. In a static approach context, using frameworks or libraries may lead to the impossibility to deduce the dependencies from this part of the application.

Example. Figure 4.1 illustrates this problem. One supposes that `methodC` has changed. The dependency chain cannot be traced back through the library. On the opposite side of the dependency chain, `testMethod` depends on the library.

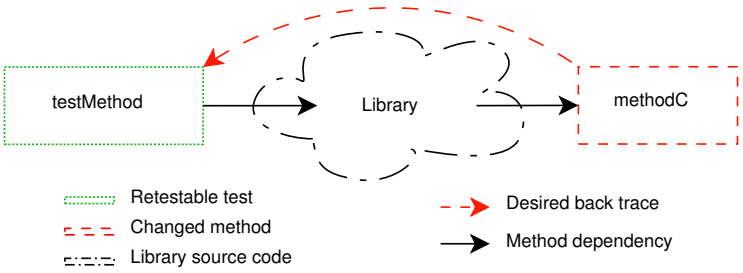


Figure 4.1: Libraries Case

Possible solution. In Java, the dependency graph can be analysed and built from the compiled code. Fortunately, Java bytecode is a somehow high-level language. We investigated with a tool implementing such a mechanism. In other languages, or for tools working on the source code, this might be a more serious issue.

1.2.2 Anonymous Classes

Context. It is accepted behavior in Java development to implement a callback mechanism through anonymous classes. For example, in GUI frameworks (Swing, SWT, Android), clicking on a button results in a call, by the framework, to a specific method of the application (callback). Very often, this method is implemented by an anonymous class, defined in another method of the application.

Example. Figure 4.2 exposes this issue. Test method `testMethod` depends on `methodA` which defines and instantiates `AnonymousClassB`. The method `anonymousMethodB` on the other hand depends on `methodC`, but this last one

is never explicitly called from `methodA`. The dependency between `testMethod` and `methodC` can only be deduced if the containment link (link between `methodA` and `AnonymousClassB`) is included in the dependency graph. For test case selection, it is not important that `methodA` never actually calls `methodC`, all that is important is that `testMethod` depends on `methodC`, a dependency that the solution can discover accurately.

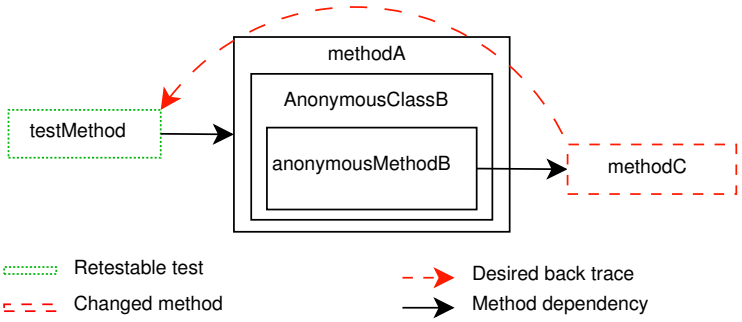


Figure 4.2: Anonymous Classes Case

Proposed Solution. It would be easy in this case to modify the static analysis tool to include containment links in the dependency graph, whether for all classes or only in the case of anonymous classes.

1.2.3 Delayed Execution

Context. This case is very similar to the previous one in its description, but the solution is different. In this issue, a class implements the `Callable` interface and, as such, implements the `call` method. This method can itself be called asynchronously according to different mechanisms (Future, Thread...).

Example. Figure 4.3 illustrates this issue. `testMethod` depends on `methodA` which depends on the `CallableImpl` constructor. `CallableImpl` implements `Callable` by implementing the `call` method. Finally, `call` depends on `methodB`. `methodA` uses an engine to perform the asynchronous task by adding the callable class and fetching the result. In this case, there is no static dependency between the `methodA` and `call`, which breaks the dependency graph.

Proposed Solution. It is possible to identify a chain of dependencies going back to a `call` method in a class implementing the `Callable` interface. The static analysis tool may be modified to prolong this dependency chain from the class implementing the `Callable` interface to the method that instantiates it. The reminder of the chain does not pose a problem and can be traced back to the tests that depend on this method.

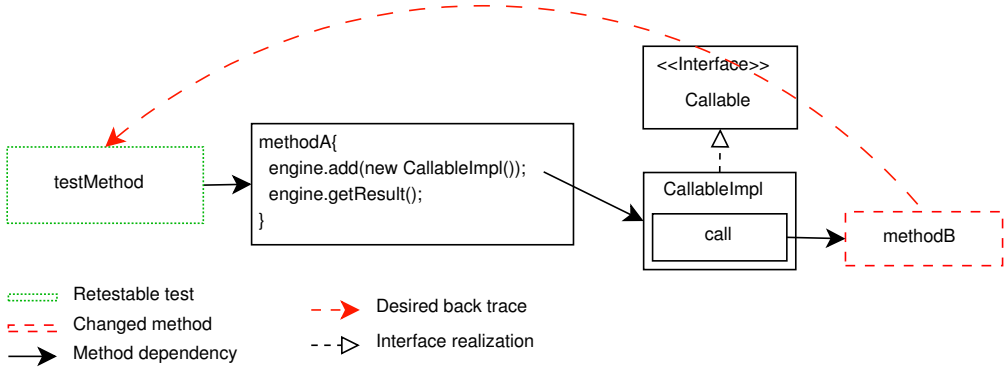


Figure 4.3: Delayed Execution Case

1.3 Multi-program Breaks

Multi-program breaks arise when two applications, from two distinct execution environments, interact. This category **impacts both dynamic and static approaches**, however, solutions for static approaches seem more practical.

1.3.1 Dynamic Calls Through Annotations

Context. Some methods are called through source code annotations. It happens, for example, when the application is composed of a client and a server side. The server side usually exposes some methods callable by the client. For example, Java J2EE defines, on the server side, objects representing database tables which can be used by a client application, through a lookup mechanism.

Example. Figure 4.4 describes the client/server problem. `testMethod` depends on `methodClient` which uses `methodServer` through a remote call. This call is possible thanks to the annotation `@Remote` defined on `ClassServer`.

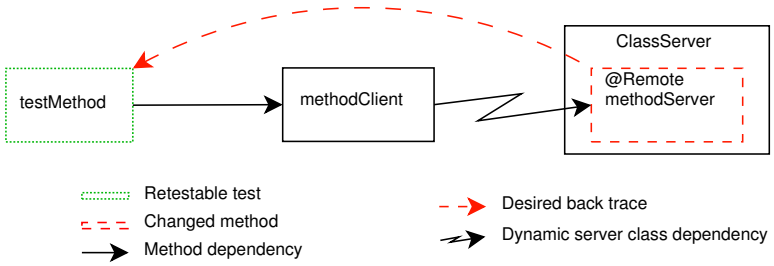


Figure 4.4: Annotation Case

Possible Solution. The mapping between the client and the server classes in a dynamic approach requires to synchronize and join the execution traces of the

client and the server for each individual test. With a static approach, there are usually markers in both applications (*e.g.,* `@Remote` on the server side) that allow to deduce the remote calls. Another possibility would be to consider a hybrid approach using both static and dynamic analyses.

1.3.2 External Tests

Context. In some cases, an external framework provides its own automated tests that call the developed application (*e.g.,* SoapUI[1]). In this case, it can be difficult to instrument the code to know which test is running and to compute its execution trace. For a static analysis approach, the problem is similar to a third-party break.

Example. Figure 4.5 illustrates the problem. An external test `externalMethod` exercises `methodB`, contained in the program under test.

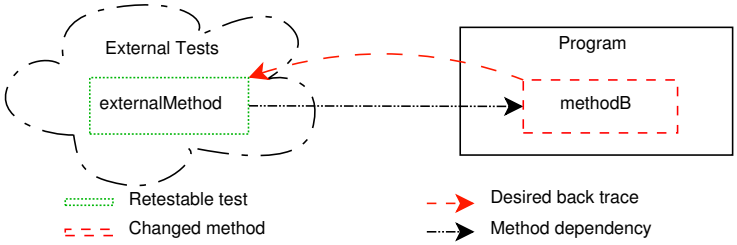


Figure 4.5: External Test Case

Possible Solution. In this case, a pure dynamic approach may not be able to solve this issue because it might be difficult to separate the execution traces of each test. Again, a hybrid approach could potentially solve this issue.

1.4 Dynamic Breaks

Dynamic breaks arise when pieces of code are treated as data with specific instructions to execute it when needed. A break might occur because the execution can be located in an entirely different location than the definition of the code. This category of issue only **impacts static approaches**.

1.4.1 Dynamic Execution

Context. Some programming languages, for example with a reflexive API like Java, allow developers to invoke code dynamically from a string.

[1] http://www.soapui.org

Example. Figure 4.6 illustrates the problem. `testMethod` depends on `methodA` which invokes dynamically `methodB`. Despite the invocation, there is no link in the dependency graph between `methodA` and `methodB`.

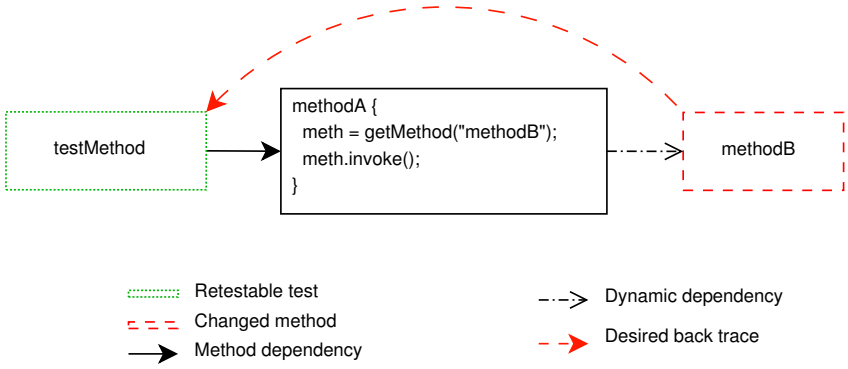


Figure 4.6: Dynamic Execution Case

Possible Solution. In this case, a pure static approach would be difficult in a generic case. However, uses of the reflexive API are very rare in the project studied.

1.4.2 Lambda-Expression

Lambda-expressions are anonymous methods. They can be passed as method parameters and therefore executed anywhere. In Java, they can be used since version 8, but many other languages also support them.

At first sight, it seems to be an issue for a static approach because the lambda-expression can be executed anywhere in the program. However, to be used, a lambda-expression has to be declared. It means that a test depending on it must first invoke the method declaring it. Therefore, we are back to normal static analysis position considering that a possible call is equivalent to an actual call.

Another view on the problem is to consider that lambda-expressions are recent in Java, and as such not present in legacy code. They are also more advanced programming artefacts that would not be found in typical information systems used in Worldline.

1.4.3 Attribute Automatic Initialization

Context. The declaration of an attribute may include its initialization through a method call. This call is performed directly in the class and not in a method scope when a new instance is created. As such, the dependency might be to the class itself and not the method called during the initialization.

Example. Figure 4.7 illustrates this problem. `testMethod` has a dependency to `methodA` which creates an instance of `ClassB`. This class defines a class attribute, named `attribute`, which is initialized with the return of `methodB`.

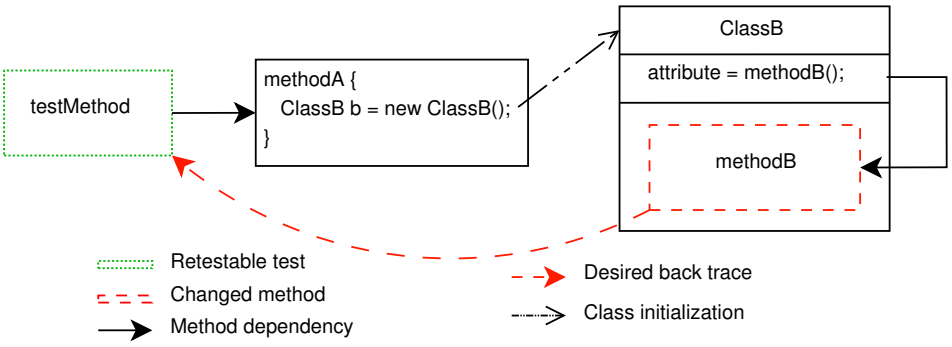


Figure 4.7: Attribute Direct Access

Proposed Solution. This kind of dynamic execution can be solved in a static way. For the classes that exhibit this problem (attribute initialization by calling a method), the creation of instances (use of `new`) should be treated as a dependence to the method.

1.4.4 Static Attribute Initialization

This issue is a variation of the previous problem where `attribute` is static. The solution is the same because the static attribute is likely initialized only when the class is actually used (through a call to `new`).

1.5 Polymorphism Breaks

Polymorphism breaks are specific to object-oriented applications. This category of issue only **impacts static approaches**.

Context. In order to specify the public methods of a class, developers often use interfaces declaring these methods. The methods of the class are not called as such but through a receiver with the interface as its type. This problem is also encountered in case of inheritance where a subclass can override a superclass method.

Example. Figure 4.8 presents the problem. `testMethod` depends on `ClassInterface.method`. And `ClassImpl.method` depends on `methodA`. `ClassImpl` implements `ClassInterface`. This implementation link is absent in the dependency graph.

Proposed Solution. The static analysis tool can be modified to include the implementation link in the dependency graph to trace back the dependency chain. In

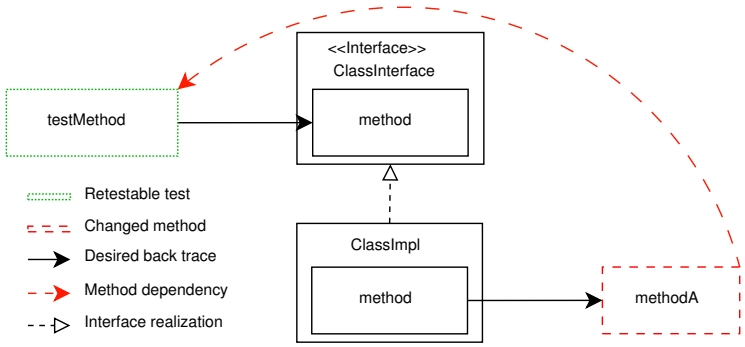


Figure 4.8: Tests Selection Approach Through Interfaces

more complex cases, *e.g.,* with several superclasses, all the links to the potential overridden methods are added to the model.

2 Experimental Setup

To ensure that our approach of test selection is adapted to the Worldline environment, we made a case study with large representative projects, written in Java. The idea is to evaluate the static approach versus a dynamic baseline, and, the impact of the various problems identified in Section 1. We used existing open-source tools that solve these problems. This section presents the tools and software projects used to carry out these case studies.

2.1 Case Study Protocol

To analyze the test selection approaches, we considered changes and navigate the model. As depicted in all the previous Figures, we navigate the model at a method level, instead of a class, instruction or package one. The idea was to strike a balance between accuracy of the selection on one hand and processing time and size of data model on the other hand.

We will not investigate the impact of the multi-program breaks (Section 1.3) because we could not compute the required baseline on which to compare to. As we saw, this category of problems impacts the dynamic approach which is used to create the baseline.

For each issue identified above, we study how its resolution impacts the test selection approach in terms of: Number of Selected Tests, Precision, Recall, and F-Measure (these metrics are rigorously defined in Section 2.4). So, the Research Questions are the following:

RQ1: What is the impact of the resolution of the third-party break issues (see Section 1.2) on the test selection approach?

RQ2: What is the impact of the resolution of the dynamic break issues (see Section 1.4) on the test selection approach?

RQ3: What is the impact of the resolution of the polymorphism break issues (see Section 1.5) on the test selection approach?

RQ4: What is the impact of combining the solutions to different problems on the test selection approach?

RQ5: What is the impact of changing the same method repeatedly (as occurs in real life) on the test selection approach?

RQ6: What is the impact of considering real commits (that change several methods jointly) on the test selection approach ?

RQ5 and RQ6 are important for the application of the approach for the company. Developers rarely modify only one piece of software at the same time. So, combining several methods and compare it to real changes is important. As real changes, we considered commits in project repositories.

All case studies follow the same pattern that we will illustrate with RQ1. To assess the impact of the third-party breaks, we will compare the results of two similar studies: one bypassing the problem, the other not. Test coverage is given by the dynamic approach (Jacoco tool) that is our baseline. The coverage from Jacoco for the baseline is close to be perfect because our version of the source code never actually changes. The baseline is only computed once. Results on the impact of the third-party breaks issue involve:

i. We fixed one version of the source code on which we work. This version never changes, all changes are virtual. No new model is created from the changed source code, but we only consider, at model level, that a method have been changed. We will ask the question "If Java method `m()` was changed, would we be able to identify the tests that cover it?" This decision was necessary because fetching the source code and the dependencies, recompiling, and running the tests for one version is resource intensive and could not be computed in reasonable time and space for such a large study.

ii. We consider as "changed" each Java method of the application that is covered by at least one test. Using a static approach, we try to identified the test cases covering this Java method.

iii. From the test cases identified, we compute different metrics (see Section 2.4)

iv. The metric values are averaged over all Java methods (covered by at least one test) to produce a result for the static approach considered.

v. The same process is repeated for all static approaches and we compare their respective results to answer the research question. The difference in the results is considered as the impact of the problem that one of the two static approaches solves.

To answer RQ1, we apply one static tool (Infinitest) that can overcome the third party break issue and another one (Moose) that cannot.

To answer RQ2, RQ3, and RQ4, we apply the same static tool (Moose) including or not the solutions to the different problems considered. For RQ4 (combining all solutions), we will not be able to include the solution to the third party break issue, because it cannot be easily done with Moose.

To answer RQ5, we apply both static tools (Infinitest and Moose) on all Java methods. The difference in the two static approaches is in the way the metric results are averaged (Step iv.). In one case, we use a weighted mean where each Java method has a weight corresponding to the number of commits (in the history of the system) where it appears in. A Java method must appear in at least in one commit (at its creation) but may be modified frequently (more than one hundred times for some cases). The weighted mean is considered more realistic as in any system, all methods are not changed with the same frequency. Thus, methods changed more frequently will have more impact on the result.

Finally for RQ6, we apply the same static tools (Infinitest and Moose) on all Java methods in one case and all system commits in the other case. As for RQ5, we use all commits in the history of the system that touched at least one method appearing in the version of the code we use. Commits differ from individual methods in that they may change many Java methods (up to 125 in one case). Commits can give better results because some Java methods with good results can "cover" for other Java methods in the same commit with unsatisfying results. For example, one Java method suffering from a polymorphism break would still see its own tests identified because another Java method not suffering from this problem was changed in the same commit. Again commits are considered more realistic than individual Java methods. We study past commits on one single code version where oracle is calculated. As commits impact several methods, we consider the union of the selected tests for each method to compute the metrics. We believe this is acceptable because they still indicate that several Java methods were changed together and we would like to know if we would be able to identify all the tests if that were the case again.

2.2 Projects

To perform our investigations, we selected three projects (P1, P2 and P3) of Worldline that we considered representative of the projects of the company in terms of

size and technologies used. P1 and P2 are financial applications with more than 400 KLOC (Kilo Lines Of Code). P1 is a service (in term of Service Oriented Architecture, SOA) dealing with card management. P2 is an issuing banking system based on SOA and reusing the card management system developed in P1 (P2 uses P1 as a third party). P3 has no relation with the two other projects, and is an e-commerce application. P2 and P3 test suites are mainly composed of integration tests, that ensure the good behavior of the application with its dependencies and the database. P1's test suite mainly includes unit tests that guarantee the results of its algorithms. In these projects, each test is a Java method using JUnit[2]. In project P1, the external code is mainly composed of libraries. Projects P2 and P3 use frameworks, including P1, making frequent use of inversion of control[3] and sub-classing. Table 4.1 gives some detailed metrics on the size of the projects and their test suites:

KLOC Core: thousands of lines of code implementing the features of the application, excluding the tests;
KLOC Test: thousands of lines of code to test the behaviour of these features;
KLOC Covered Core: thousands of lines of code covered by at least one test;
#Green Tests: number of tests that are green on the version of the application considered for our investigations;
#Method: total number of methods in the application;
#Covered Methods: number of methods with at least one line of code covered;
Avg Methods/Test: average number of covered methods by test;
Avg Tests/Methods: average number of tests covering a method which is covered by at least one test;
#Commits: number of commits for each project;
Avg Methods/Commit: average number of core methods changed by commit;
Repository Creation: year of creation of the repository (remember however that we only consider commits touching a method that still exist in the fixed code version).

P1, P2 and P3 are big applications (hundreds of KLOC). P1 includes 5,323 valid tests; P2, 168; and P3, 3,035. In P1, the tests cover 4,720 methods (48%), in P2, only 3,261 methods (6%), and in P3, 8,143 methods (18%). One could think that these are rather low coverage values (particularly P2). Two facts may explain this. First, the projects are old. They date from a period when there was a less strong emphasis on automated testing in the company and manual testing was prevailing. Second, we could not use all the tests, either because they were based on specialized tooling that we could not instrument (see Section 1.3.2), or because they failed. For these projects, only the tests handled by the dynamic

[2]http://junit.org/
[3]The client does not call the framework but the framework calls the client.

Table 4.1: Global metrics of projects P1, P2 and P3

Metric	P1	P2	P3
KLOC Core	447	716	302
KLOC Tests	184	48	74
KLOC Covered Core	97	49	74
#Green Tests	5,323	168	3,035
#Method	9,808	56,661	45,671
#Methods Covered	4,720	3,261	8,143
Test Coverage	48%	6%	18%
Avg Methods/Test	83	134	152
Avg Tests/Method	54	2	35
#Commits	2,217	467	2,115
Avg Methods/Commit	24	129	37
Avg Files/Commit	7	18	17
Repository Creation	2009	2015	2009

approach, are considered for the study, *i.e.,* those that run without crashing. We were surprised to find that some tests in each project were failing outright. This impedes to use the dynamic approach (no execution trace) and thus to use them in our studies. According to Pinto et al. [2012], tests that fail in such a new version tend to be deleted not because they are difficult to repair, but because they are obsolete. In their study, the authors found that failing tests are more often deleted (1,594 instances of deletions, 59%) than repaired (1,121 instances of repairs, 41%).

The values of the KLOC Covered Core and #Green Tests metrics, in Table 4.1, only take these tests into account. P1 and P3 are 6 years older than P2, with respectively 2,217, 467, and 2,115 commits that we considered. P2 is actually a rework on some legacy code dating back from 2010. However, we were not able to recover the commits from the early version of the project. These projects are still alive and evolving regularly.

Test execution (compilation and test execution included) requires 3 hours for P1; 2 hours for P2; and 30 minutes for P3. This only includes the tests that we are considering in our studies (*i.e.,* excluding broken tests). This time is mainly due to the setup of each test (database population, server startup and configuration); test data volume; and the fact that there are abnormal conditions tests (timeout).

Commits seem rather big for P2, there are more than 100 methods, and 18 files by commit.

2.3 Dynamic and Static Approaches Tooling

For this study, we used the open-source tools defined in Section 2.2. All these tools had to be modified to allow comparing them in our study. They deal with the whole program at once whereas we want to evaluate the selection change by change.

For Jacoco, we modified the tool in order to separate information relative to each test and thus know which test covers which Java methods. An aspect has been implemented to start the recording at the beginning of each test and serialize the coverage results at the end. The method dependencies associated to each test are thus available.

Infinitest has been modified to be used in a standalone mode rather than through an IDE, and, in Moose, approaches for test selection has been implemented on its static model.

2.4 Metrics

An optimal test selection approach selects the smallest set of all test cases covering a change. Biswas et al. [2011] use two characteristics of the test case selection approaches to evaluate if they are *safe* and *precise*. An approach is *safe* if it selects all the tests that reveal a modification. An approach is precise if it selects only tests that reveal modifications of source code and not other tests. Therefore according to these definitions, all test case selection techniques will exhibit some level of imprecision, the tests selected can traverse a changed method but may not reveal its modification. A safe approach has a recall of 1 and a precise approach has a precision of 1.

According to the case study protocol (Section 2.1), the methods are not changed but considered as such each in turn. As a consequence, the dynamic approach is both precise and safe: all the tests selected by the dynamic approach cover the changes; no other test covers a given change.

Due to the issues we identified, static approaches may miss some tests covering a change and select others that do not cover it. To compare the approaches, we focus on the quality of the test selection approach using four metrics that can be computed from the traditional quantities:

- **True Positives** (TP) is the number of tests selected by the static approach (study) and the dynamic approach (baseline);
- **False Positives** (FP) is the number of tests selected only by the static approach;
- **False Negatives** (FN) is the number of tests selected only by the dynamic approach;
- **True Negatives** (TN) is the number of tests selected neither by the dynamic

nor by the static approaches.

From these quantities we compute the following metrics: *Selected tests*, *Precision*, *Recall*, and *F-Measure*.

Selected tests represents the number of tests selected by the approach as a ratio of the number of selected tests to the total number of tests. This metric corresponds to $1 - cost\ reduction\ criteria$. The cost reduction criteria [Engström et al., 2010] computes the decrease in the number of tests. A cost reduction criteria value close to 1 means that the number of tests to relaunch is small compared to all the tests of the application. On the other hand, a value close to 0 means that the approach selects almost all the tests.

$$Selected\ tests = \frac{TP + FP}{TP + FN + TN + FP}$$

Precision is the fraction of retrieved tests that are retestable (see Section 1.1). A high precision means that the static approach selects essentially tests that cover the changed method.

$$Precision = \frac{TP}{TP + FP}$$

Recall is the fraction of retestable tests that are retrieved. A high recall means that the approach is safe and that the tests covering a given changed method are selected by the static approach.

$$Recall = \frac{TP}{TP + FN}$$

F-Measure is the harmonic mean of *Precison* and *Recall* to show the overall performance of an algorithm.

$$F\text{-}Measure = \frac{2 * Recall * Precision}{Recall + Precision}$$

Priority will be given to a higher *Recall* to make sure one does not overlook a test that could discover a bug in the changed code. However, achieving good recall is easy by selecting many tests. This would show up in the *Precision* and *Selected tests* metrics. We must remember that our goal is ultimately to be able to give rapid and useful feedback to the developer right after he commits a change. So, better *Precision* would mean no useless test run. But *Precision* typically comes at the expense of *Recall*, so good *Precision* could also mean some needed tests would not be run, and, therefore, we could not give any guarantee to the software engineers about the quality of their development. This would defeat the purpose of testing the application.

Furthermore, as it is, our investigations show than we select a very small percentage of the entire test suite often less than a hundred tests. In these circumstances, we give preference to *Recall* to ensure more accurate feedback.

3 Results and Discussion

This section presents our case study results to answer the six research questions. Table 4.2 gives the metrics for each static approach. Figure 4.9 represents the boxplots based on the last line of Table 4.2.

3.1 RQ1 – Third-Party Breaks Impact

To answer this Research Question, we consider Infinitest (that overcomes this issue) and Moose (that does not). Because Infinitest works at the class level, we do the same for Moose. Two characteristics of Infinitest had to be replicated in our study to allow comparison with this tool. First, Infinitest works at class level, dependencies in the code are projected at class level (*e.g.,* an invocation between two methods is raised up as a dependency between the classes containing these methods). Second, Infinitest uses class imports to select tests, consequently, direct inheritance, invocations, accesses, and references to classes are taken into account. We created a "Moose for Infinitest" version where we raise dependencies at the class level and follow the inheritance, access, and reference links in addition to the invocation links.

Infinitest has higher *Selected tests* for the three projects, but the difference with "Moose for Infinitest" is low. For P1 and P3, the *Precision* is worse and the number of tests selected decreases and P2 behaves as would be expected (higher *Precision* when there are less tests selected). There might be some special condition on P1 and P3 that makes them behave this way. One explication is the presence of more inheritance or accesses than in P2. P2 extends the classes of P1 to implement specific features.

Recall is better for Infinitest for the three projects (resp. 72%, 66%, and 44%) which is normal since it selects more tests. The difference however is small as will be seen when looking at the next metric.

F-Measure is more consistent and gives better results for the three projects to Moose (not solving the third party breaks).

Based on the *F-Measure* results, one could conclude that there is no urgent need to solve the Third Party Break issue on our three projects. The *Recall* results are slightly lower (from 72% to 70% for P1; from 66% to 63% for P2; and from 44% to 41% for P3), but this could be acceptable since the time to run the selected tests will be shorter.

3.2 RQ2 – Dynamic Breaks Impact

For this Research Question, three Moose approaches, at method granularity level and that bypass different dynamic break issues are investigated. They are to be

Table 4.2: Comparison of the static approaches to the dynamic one for test case selection considering all Java methods individually

		Selected Tests			Precision			Recall			F-Measure		
		P1	P2	P3	P1	P2	P3	P1	P2	P3	P1	P2	P3
Jacoco (dynamic)		0.8%	1%	0.4%	-	-	-	-	-	-	-	-	-
Infinitest	RQ1	23%	5%	3%	9%	39%	15%	72%	66%	44%	12%	43%	18%
Moose for Infinitest		19%	2,2%	2%	10%	27%	18%	70%	63%	41%	13%	44%	20%
Moose (methods)	Base	0.4%	0.1%	0.1%	43%	11%	24%	36%	11%	13%	35%	11%	15%
Moose w/ delayed exec.		0.4%	0.1%	0.1%	43%	11%	24%	36%	11%	13%	35%	11%	15%
Moose w/ anonym. classes	RQ2	0.4%	0.1%	0.1%	43%	11%	24%	36%	11%	13%	35%	11%	15%
Moose w/ attributes		0.4%	0.2%	0.1%	43%	17%	24%	36%	17%	13%	35%	17%	15%
Moose w/ polymorphism	RQ3	3%	0.4%	2%	43%	25%	34%	91%	26%	41%	50%	25%	29%
Moose w/ att. & anon. & polym. & delayed exec.	RQ4	3%	0.8%	2%	43%	61%	34%	91%	64%	41%	50%	62%	29%

compared with the Moose approach at method granularity level (line "Moose (methods)" in Table 4.2).

For the three projects, we see almost no change between Moose solving one of the specific Dynamic Break issues and Moose not solving any issue. The only exception is slightly more *Selected tests* for P2 when solving the Attribute Automatic Initialization (Section 1.4.3) issue, followed by significantly better *Precision*, *Recall* and consequently *F-Measure*.

The first conclusion would be that it is mostly useless to try to solve these issues. We will see however in Section 3.4 that issues may be intertwined and that solving only one alone might not be enough.

3.3 RQ3 – Polymorphism Breaks Impact

For this Research Question, the Moose approach at method granularity is compared to the Moose approach (at the same granularity) bypassing the polymorphism issue.

The three projects have more *Selected tests* in the Moose with Polymorphism approach than in the base approach. *Precision* improves significantly for P2 and P3 and remains equal for P1. Again an improvement here is unexpected since we selected more tests. *Recall* improves drastically for P1, from 36% to 91% and significantly for P2 and P3. And of course *F-Measure* improves also for the three projects.

The conclusion is that it was very important to solve this specific issue in our cases. P1 particularly shows excellent results, with $> 90\%$ *Recall*, a still good *Precision* (43%, about half of the selected tests do cover the changed method) and a similarly good rate of *Selected tests*. This can be explained by the high usage of Java interfaces in the projects. Interfaces are used to express the methods of the class callable from outside frameworks.

3.4 RQ4 – Impact of Combining Solutions

For this research question, we look at Moose approaches combining solutions for anonymous classes, delayed execution, polymorphism, and dynamic break issues. This will be compared to the bare Moose approach at method granularity with no solution implemented, as well as other Moose approaches with any individual solution included.

The combination of all implemented solutions gives very good results. Overall, the results for P1 and P3 are similar to the ones of the previous study (RQ3, Section 3.3) and P2 is showing more *Selected tests*, much better *Precision*, *Recall*, and *F-Measure*.

Results for P2 show that combining solutions to several issues at the same time improves the situation in a way that would not be expected if one looks at the

individual results of each solution. The conclusion we draw from this is that issues are intertwined and must be solved jointly.

Another conclusion is that by answering all the issues (minus the Third Party Breaks that Moose cannot solve easily), we end up with very good results, *Precision* ranges from 34% (P3) to 61% (P2), and *Recall* ranges from 41% (P3) to 91% (P1). These results position the static approach as a viable solution to the test case selection problem.

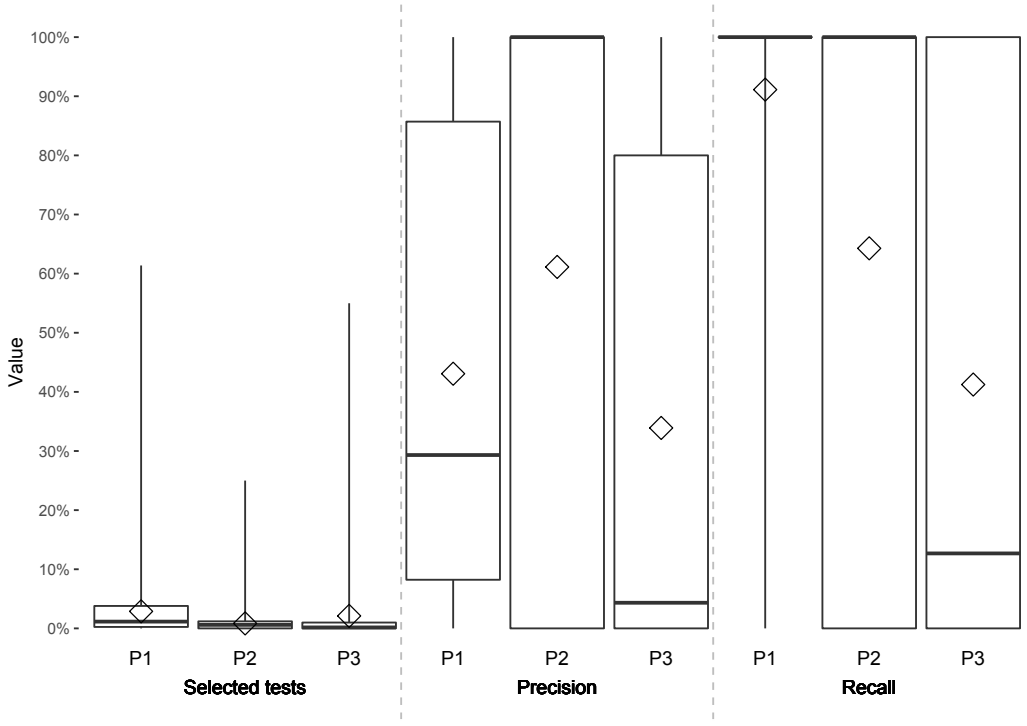


Figure 4.9: Boxplot of the distribution of the *Moose w/ att. & anon. & polym. & delayed exec.* study considering all Java methods individually. The diamonds represent the mean value of the metric (presented in Table 4.2)

Figure 4.9 shows as boxplots the distribution of the data of our last study considering all java methods independently. For example, the boxplot for the *Precision* of P1 means that the minimum reached is 0%, the maximum is 100%, the median is around 30%, and first and third quartiles are close to 10% and 85%. The mean is represented with a diamond. On the Figure, the distribution of the *Precision* of P2, and the *Recall* of P2 and P3 is *"binary"*. Almost all the values are equal to 100% or 0% (only in 25% of the cases, *Precision* (resp. *Recall*) has an intermediary value). This can happen, for example, with methods covered by only one test: either the

test is found by the approach (*Recall* = 100%) or not (*Recall = 0%*). For *Precision*, this happens, for example,when only one test is impacted after a modification: either it was selected (*Precision* = 100%) or not (*Precision* = 0%). In this case of *"binary"* results, the median is of less value because it will be either 100% or 0%. Here, the mean gives a better estimation of the central value of the results.

Sometimes, the static approach selects up to 62% of the test suite. This behavior can happen if a method that is often called is modified.

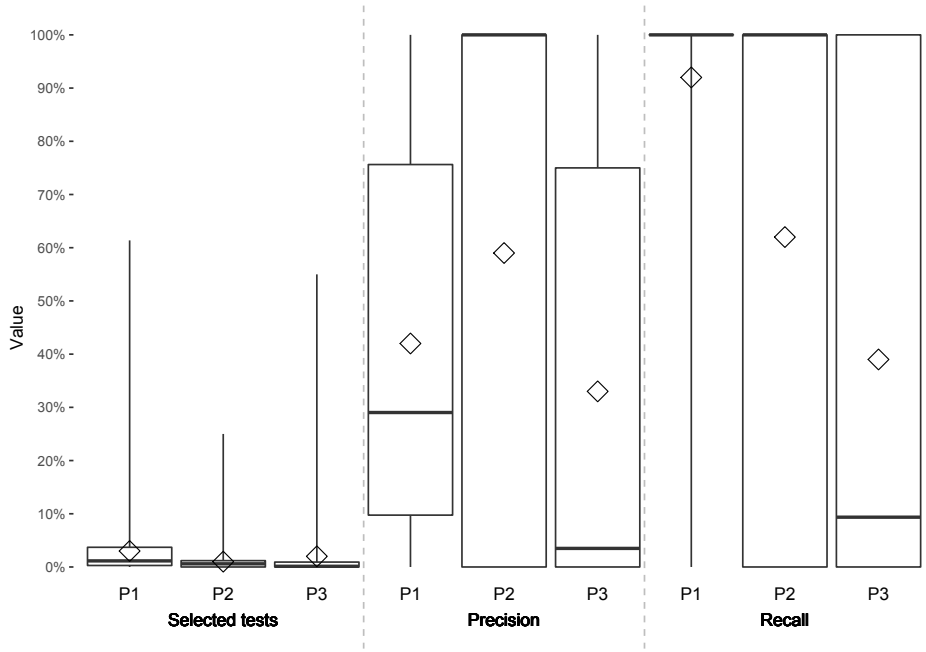


Figure 4.10: Boxplot of the distribution of the *Moose w/ att. & anon. & polym. & delayed exec.* study considering a weighting of Java methods with the number of commits they appear in

3.5 RQ5 – Weighting of Results with the Number of Commits

For this study, we summarized the results in a new table (Table 4.3) that should be compared to the first one. Figure 4.10 represents the boxplots based on the last line of Table 4.3. Although we give the results of all studies for the sake of completeness, we will be focusing on the last line in our discussion. In these new studies, the results of each methods are weighted according to the number of commits the method appears in (Section 2.1). The idea is that the most committed Java methods could have consistently good (or bad) results.

Table 4.3: Comparison of the static approaches to the dynamic one with a weighting of Java methods with the number of commits they appear in

		Selected Tests			Precision			Recall			F-Measure		
		P1	P2	P3	P1	P2	P3	P1	P2	P3	P1	P2	P3
Jacoco (dynamic)		1%	2%	1%	-	-	-	-	-	-	-	-	-
Infinitest	RQ1	18%	5%	4%	7%	35%	14%	59%	61%	39%	9%	40%	16%
Moose for Infinitest		15%	2%	2%	9%	28%	17%	57%	59%	38%	11%	40%	19%
Moose (methods)	Base	0,3%	0,2%	0,2%	36%	13%	23%	27%	12%	12%	27%	12%	13%
Moose w/ delayed exec.		0,3%	0,2%	0,2%	36%	13%	23%	27%	12%	12%	27%	12%	13%
Moose w/ anonym. classes	RQ2	0,3%	0,2%	0,2%	36%	13%	23%	27%	12%	12%	27%	12%	13%
Moose w/ attributes		0,3%	0,2%	0,2%	36%	16%	23%	28%	17%	12%	27%	16%	13%
Moose w/ polymorphism	RQ3	3%	0,4%	2%	42%	27%	33%	92%	28%	39%	50%	27%	28%
Moose w/ att. & anon. & polym. & delayed exec.	**RQ4**	**3%**	**1%**	**2%**	**42%**	**59%**	**33%**	**92%**	**62%**	**39%**	**50%**	**60%**	**28%**

In summary, the results (last lines of Tables 4.2 and 4.3) are not very different. This new study consistently brings marginal decrease in *Precision* and *Recall* and small increase in *Selected tests*. Since the value for one given method is the same in both studies, the difference, on average, can only come from the weighting of the methods results and, therefore, it seems that "bad" methods would have higher weight.

This would suggest that the methods where static approaches only select the wrong tests are more frequently committed. This is not good news, but the differences are small (typically one percentage point) and would need to be more formally tested in a specific case study. Moreover, Figure 4.10 results are almost the same as those presented in Figure 4.9 which reinforces the small impact of the changes.

3.6 RQ6 – Aggregation of the Results by Commit

To answer this last Research Question, we again replicate all studies, but working with commits instead of individual methods. All the results are summarised in Table 4.4 but we will concentrate on the last line and compare it to the one in Table 4.2. Figure 4.11 represents the boxplots based on the last line of Table 4.4.

The first observation regards the number of *Selected tests*. Since commits comprise many Java methods (average for P1 is 24, for P2 it is 129, see Table 4.1), it is expected that more tests would be selected. This is the case with our baseline (Jacoco) with a larger percentage of all tests selected (P1, from 0.8% to 8%; P2, from 1% to 21%; P3, from 0.4% to 14%). This is an increase in the range of an order of magnitude. However, we see that the static approaches (mainly at the method granularity level) tend to exhibit a smaller increase in the number of selected tests (P1, from 3% to 4%; P2, from 0.8% to 3%; P3, from 2% to 6%). So even-though the static approaches selected more tests, one could conclude that they are actually more selective than necessary here: P1 and P3 improve their *Precision* (resp. from 43% to 55%; and from 34% to 49%), but P2 decreased (from 61% to 45%). So good news for P1 and P3, that are more selective but also more precise.

Being more selective, the *Recall* results were bound to worsen: the approaches select less tests than they should according to our baseline. This is what happens with P1 and P2 (resp. from 91% to 81%; and from 64% to 45%), but P3, which previously had the lower *Recall*, improved it (from 41% to 56%).

One conclusion is that it does not seem to be the case that one Java method in a commit "covers" for another one. That might indicate that the commits touch various concerns for which different subsets of tests are necessary. That would be coherent with the large size of the commits that we already mentioned.

Moreover, results presented in Figure 4.11 are different than those of Figure 4.9. The distribution of the Precision tends to increase while the Recall decreases ex-

Table 4.4: Comparison of the static approaches to the dynamic one to test case selection, considering Java methods grouped in commits

		Selected Tests			Precision			Recall			F-Measure		
		P1	P2	P3	P1	P2	P3	P1	P2	P3	P1	P2	P3
Jacoco (dynamic)		8%	21%	14%	-	-	-	-	-	-	-	-	-
Infinitest	RQ1	13%	8%	7%	25%	26%	33%	62%	51%	62%	28%	30%	35%
Moose for Infinitest		9%	3%	5%	10%	36%	28%	30%	42%	44%	11%	28%	28%
Moose (methods)	Base	0,8%	0,7%	0,9%	54%	30%	63%	36%	21%	33%	39%	23%	38%
Moose w/ delayed exec.		0,8%	0,7%	1%	54%	30%	63%	36%	21%	33%	39%	23%	38%
Moose w/ anonym. classes	RQ2	0,8%	0,7%	1%	54%	30%	63%	36%	21%	33%	39%	23%	38%
Moose w/ attributes		0,9%	0,7%	1%	54%	30%	63%	36%	21%	34%	39%	23%	38%
Moose w/ polymorphism	RQ3	4%	2%	6%	55%	34%	49%	81%	31%	56%	56%	29%	42%
Moose w/ att. & anon. & polym. & delayed exec.	RQ4	4%	3%	6%	55%	45%	49%	81%	45%	56%	56%	40%	42%

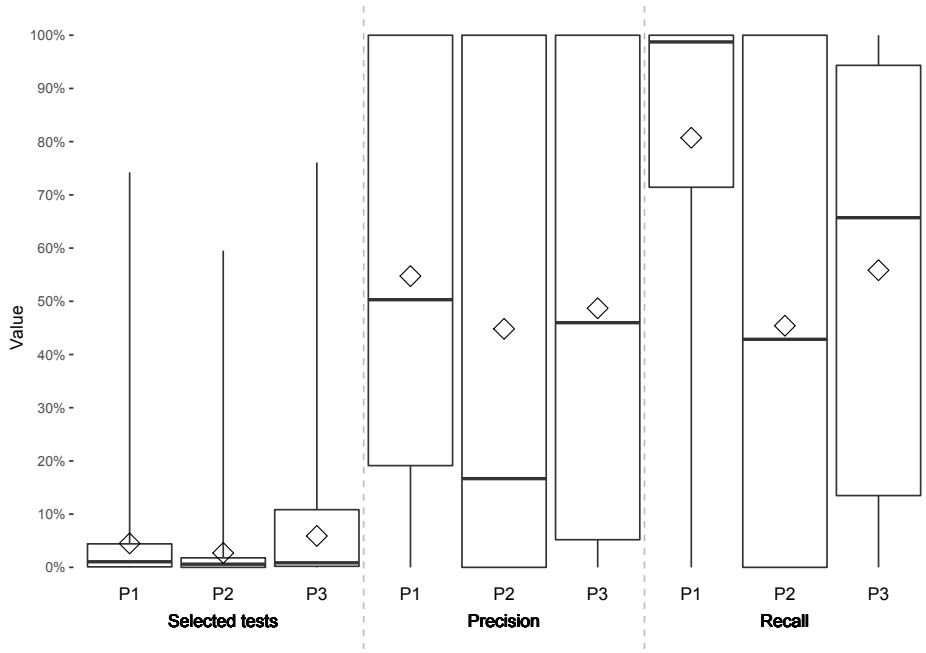


Figure 4.11: Boxplot of the distribution of the *Moose w/ att. & anon. & polym. & delayed exec.* study considering Java methods grouped in commits

cepted for P3.

Considering commits instead of individual Java methods also means that Java methods are weighted, just like in the preceding study. We saw a very small worsening of the results (more tests selected and recall decreasing) when weighting the Java methods and this could account for a part of the bad results here: it is possible that Java methods with bad results are committed more often than the other ones.

3.7 Overall Conclusions

We draw three overall conclusions from these investigations.

First, issues highlighted in Section 1.1 might be intertwined. This means that finding the tests from a given method can exhibit several of the issues we identified. This implies that solving only one of the issues would not allow to identify all the tests that cover this method. Intertwining may be influenced by the development patterns used in the application.

Second, issues do not have the same impact on the projects. This might be the consequence of different coding conventions or rules. For example the attribute initialization issue (Section 1.4.3) is not present in P1. In contrast, the polymorphism issue has a high impact on P1's results. In P1, external libraries are largely used.

In Java such uses occur through calls to implementation and interfaces. Dynamic breaks only impact P2 due to the specificity of the application to make batches of operations. Such issues might be helped by establishing better coding conventions.

Third, considering commits instead of individual Java methods tend to worsen the results with approaches that are too selective to keep the same level of good results. The large size of the commits might be an important factor in this behaviour.

Another conclusion from Section 1.1 would be that even in one category, issues can unfortunately be very different and require each a specific solution.

4 Evaluation of Validity

This section discusses the validity of our case study using validation scheme defined by Runeson and Höst [2009]. The construct validity, the internal validity, the external validity, and the reliability are presented.

4.1 Construct Validity

Construct validity indicates whether the studied operational measures really represent what is investigated according to the research question. The purpose of this study is to evaluate pros and cons of different approaches for test case selection and compare static and dynamic approaches to select the optimal test set to execute after a change in the application.

Metrics Validity

Four metrics have been chosen: the ratio of selected tests, the *Precision*, the *Recall* and the *F-Measure*. These metrics are considered relevant in Biswas et al. [2011] and Engström et al. [2010]. There might be an issue with our cost reduction metric which does not consider execution time but number of test selected.

Another point related to the time gain that one may hope from test selection is linked to the initialization of the tests. Indeed, a part of the total time to test is caused by long initialization. This initialization can occur for the entire test suite, for each test-class (annotation `@BeforeClass` in JUnit 4) or each test-method (annotation `@Before` in JUnit 4). Selecting tests has the potential to reduce the `@BeforeClass` and `@Before` initializations. We performed a small study with one commit that led to 43 test-methods selected (on a total of 5,323) in 7 test-classes (on a total of 392). Resulting testing time was 13 min: 4 min for the first test-class (including global initialization + `@BeforeClass`) and 1 to 3 minutes for each successive test-class (including only their respective `@BeforeClass`). Based on this result, we assume that, for this project, test selection will allow to

reduce testing time because all initializations are not done globally but also on a test-class or test-method basis.

Granularity Level Validity

For these case studies, we mainly considered a method granularity. We used class granularity only for RQ1 where our comparison basis was at class level.

By comparing both approaches, the cost reduction for Moose/class is much worse than Moose/method. The precision is also worse, but, recall improves. For P1, the percentage of selected tests goes from 19% in the case of Moose/class to 0.4% in the case of Moose/method. For P2 and P3, these percentages are respectively around 2% in the case of Moose/class and 0.1% in the case of Moose/method. Moose/class execution time is around 34 minutes for P1 (down from an original 3 hours), and around 3 minutes for P2 and less than one minute for P3. These execution time values are good. As we want to have a high *Recall* and a small set of tests selected, we have yet no clear argument to choose the best granularity level. On one hand, as exposed before, the time required to initialize tests is a concern. Part of this initialization is performed once for each test class. For this part, working at class level would be a good choice as there would be little additional gain in term of time from selecting only some test methods. But this kind of generalization cannot be done on all projects of Worldline. It is also possible that each test sets up its own environment. In this case, running all the tests of the chosen class will make developers loose time by launching un-relevant tests (precision for Moose/class is worse than for Moose/method).

4.2 Internal Validity

Internal validity indicates whether no other variables except the studied ones impacted the result.

There was an issue with anonymous classes that, obviously, cannot be identified by their name from one study to the other. For this reason, their methods have not been considered as "changed" in the studies.

We have been careful to test every solution independently of the other before combining them. Our modifications of Infinitest and Jacoco did not introduce unwanted errors in the results of these two tools as we did not touch the algorithm parts but how they are run on the tests.

Finally, Tengeri et al. [2016] argue that Jacoco which is based on bytecode instrumentation may produce erroneous results compared to source code instrumentations methods. Jacoco misses some really covered methods. At first, the difference in the overall percentage of both approaches presented by Tengeri et al. might not seem too big (between 0.5 – 8.5%), but relative to the actually covered

elements, the difference can be as high as 24%. However, on a per-test and per-method levels, it can have bigger impact. According to Tengeri et al. [2016], the delta on the coverage can be explained by another way to handle the project submodules, instrumentation issues, or errors in name encoding. By using the same tool, the confidence in the results stays the same whatever the tests covering a method: the instrumented bytecode will result in the same executed code, the handling of the project submodules, and the name encoding will stay the same. This issue may impact precision and recall of a method only if it is supposed as marked by Jacoco and the coverage of one test that should be related to this method, is not detected by the tool.

4.3 External Validity

External validity indicates whether it is possible to generalize the findings of the study.

We are fully aware that our results cannot be easily generalized as such and took precautions to present them in their true context. The results presented in this paper involves only three Java projects of one company. These three projects are different considering their size, number of tests, test coverage, and used frameworks and annotations. Moreover, they face different issues (*e.g.,* P1 does not present the attribute initialization issue).

Some of the issues discussed in Section 1.1 are fully independent of the programming language: such as external source code, testing exercised by external applications, or dynamic execution. Other issues, such as anonymous classes, delayed execution, dynamic call through annotations, attribute automatic initialization, are specific to Java.

5 Comparison to Other Works

Two other studies [Ekelund and Engström, 2015, Soetens et al., 2013] implement tests selection approaches and provide the same metrics as for our case study (selected tests, precision, and recall). Table 4.5 gathers the results of these studies.

Soetens et al. [2013] propose a static approach at method granularity based on the FAMIX meta-model. Their approach relies on real change sets gathered in commits. For each of these change sets, their static approach is compared to a dynamic one used as reference. The dependency graph they use only contains links between methods. Two open-source applications (Cruisecontrol and PMD) are used as input data for their study. For both applications, 1% of the test cases is selected. They obtain respectively, for each application, a recall of 77% and 58%, and a precision of 84% and 83%. These results are better than our Moose/method approach. First,

Table 4.5: Comparison of the static approaches to the dynamic one to select the tests after a method change

	Project/Approach	Selected Tests	Precision	Recall
Our Approach	P1	4%	55%	81%
	P2	3%	45%	45%
	P3	6%	49%	56%
Soetens et al.	PMD	1%	83%	58%
	Cruisecontrol	1%	87%	77%
Ekelund and Engström	Wide approach	37%	1.5%	95%
Ekelund and Engström	Narrow approach	4%	7.4%	79%

P1 and P2 are larger projects than PMD and Cruisecontrol that counts around 20 times less lines of code. Second, P1 and P2 use a lot of frameworks and libraries which does not seems to be the case for PMD and Cruisecontrol. However, by solving all the previously identified issues, we obtain results close to Soetens et al. in terms of recall: despite our precision being much lower (close to 30% less), we achieve a better recall for P1 with 81%. So, resolving the issues seems to improve the recall, to select more tests and to lower the precision. Our approach is coherent with the one of the authors and could be used by the developers of Worldline.

Ekelund and Engström [2015] select tests based on test result history. This history archives changes at package granularity and corresponding test results for each build of the application (*i.e.,* the execution of all the test cases by a continuous integration server). Such an approach has been defined since no other existing approach based on source code, bytecode, or dynamic analysis was possible due to the huge size of the studied application that counts several million of lines of code. When a package changes, thanks to history data mining, the authors know the potentially affected tests and select them. These tests are the ones that failed at least once when, in the past, this package changed (narrow approach) or whatever the package changed (wide approach). The accuracy of the selection algorithm is related to the number of builds used. However, considering a too large history may introduce noise in the selection mechanism since the source code may have evolved a lot. The authors found that the algorithm is optimal for a history containing 100 builds. This approach is language independent and uses few resources but relies on a history of the build results. Such a history does not often exist in companies and requires time and effort to be built. In the case of the wide approach, the ratio of selected tests reaches 37%, the precision and the recall are respectively 1.5% and 95%. In the case of the narrow approach, only 4% of the tests are selected with a precision of 7.4% and a recall of 79%. The studies of Ekelund and Engström lead to recall with the same order of magnitude than our project P1. However, precision

results are very low, because they work at package level. In Worldline, such an approach is complex to put in place: yet, no history of the test execution is recorded. Having the test history for all the projects of the company is costly because, first, some the projects tests are not launched on integration servers, second, the maintainers of these servers do not like to see such an history stored: it congests the hard drives.

6 Conclusion

To reduce the number of tests the developers have to run after a change in their source code, we investigated different approaches on three Java projects of the company. These projects counts several thousands of lines of code. For adapting the test selection approach to Worldline developers, we met several issues that we generalized and categorized. Solutions to these issues were also proposed and implemented.

From the case studies we carried out, we draw three conclusions: First, the issues we discovered might be intertwined which means that a given method can exhibit several of the issues we identified. As a consequence, solving only one of the issues would not allow to identify all the tests that cover this method. Several solutions need to be combined together to fully resolve any of the issues. Intertwining may be influenced by the development patterns used in the application. Second, problems do not have the same impact on the projects. Despite that they have a common ground, each project deviates from the common guidelines and uses its own frameworks. It can influence the test selection. However, in Java, the polymorphism problem is recurring and solving it has a great impact on the test case selection results. And third, even in one category, issues can unfortunately be very different and each requires a specific solution. For example, Polymorphism Break requires to find the superclasses of a modified class whereas Attribute Automatic Initialization Break requires to search for uses of instance variables.

Among the considered approaches, the dynamic one has been considered an accurate baseline. However, this approach has two major drawbacks: First, it is not generic and so depends strongly on the data used for the tests. Second, if a test is failing, it cannot be selected by this approach.

To put in practice the test selection, the static approach with all the known drawbacks resolved seems to give results close to the dynamic one. Moreover, it brings some advantages: tests do not have to be launched to establish a baseline at the beginning of the test selection, and, to ensure the good selection of the tests during the modifications of the source code.

CHAPTER 5

Study of Developers' testing behavior in a Company

Contents

Reports on developers behavior about testing in the literature involve highly distributed open-source projects, or are based on a study of students programmers. As a company might behave differently, we want to enhance our comprehension of its environment. A secondary purpose is to collect base data to detect the possible impact of future automated test selection actions. To fulfill these goals, we studied Worldline developers by taking inspiration from experiments of two papers of the literature [Beller et al., 2015, Gligoric et al., 2014], described in Chapter 3. Both studies establish a baseline to compare to. Our following field study describes how often the developers use tests in their daily practice, whether they use tests selection and why they do it or not. Results are reinforced by interviews with developers involved in the study.

1 Experimental Setup

This section presents the research questions that we set and the methodology to answer them.

1.1 Research questions

The two experiments in Beller et al. [2015], Gligoric et al. [2014] seem to answer well our need of studying the habits of developers: the first paper characterizes how developers use tests in their daily work, and the second one characterizes how developers can use RTS to provide faster feedback after modifying a piece of code.

However, both experiments also had characteristics that did not fit well into our context and that might render their conclusions useless to us. Neither study is made in an industrial context: Beller et al. [2015] use open-source, students, and industrial developers, and, Gligoric et al. [2014] use students and industrials. From these studies, we only kept the questions of interest. We set the following research questions for our case study:

RQ1: How and Why Developers Run Tests?

This research question mostly take inspiration from both papers. It is decomposed as follows:

RQ1.1 Do developers test their code changes?

RQ1.2 How long does a test run take?

RQ1.3 Do quick tests lead to more test executions?

RQ1.4 Do developers practice test selection?

RQ1.5 What are common scenarios for manual RTS?

RQ2: How Do Developers React to Tests Run?

This research question stems from the paper of Beller et al. [2015]:

RQ2.1 How frequently tests pass and fail?

RQ2.2 How long does it take to fix a failing test?

RQ3: How and Why Developers Perform Test Selection?

This last research question includes most of the questions from the paper of Gligoric et al. [2014]:

RQ3.1 Does manual test selection depend on size of test suites?

RQ3.2 Does manual test selection depend on size of code changes?

RQ3.3 How does manual test selection compare with automated one, in terms of precision and safety?

1.2 Experimental protocol

Participation to the case study was voluntary. We advertised it on the internal mailing of Worldline and we set up a lottery to attract more volunteers. We also did as much advertising as possible through our network of relations. Participants had to download and install the plugin we developed (see after). The plugin made data collection completely transparent for the participants which was a strong requirement for them. We also kept anonymous the data collected. This way, participants know that the managers cannot retrace the developers who do not run enough tests.

For data collection, we needed information on the test runs and from the source code (to compute code changes). One difficulty is that the development environments of the company are heterogeneous. Developers can code in the IDE of their liking (usually Eclipse or IntelliJ), and use different frameworks to run their tests (usually JUnit or Maven). The versions of all these tools are also not always the same.

Developers were *very* concerned that the participation to the case study should not add any burden or delay to their normal work. This, combined with the heterogeneous aspect of the developers environments, limited the data we could collect. It made it very difficult to log data with the same level of detail as the two papers we compare to (down to keyboard and mouse events for Beller et al. [2015]). This in turn impacted how well we could answer some questions (see Section 1.3). We collected test information through a plugin that was developed for Eclipse and for IntelliJ. It logs the same data:

Developer id: A unique id given to the developer;
Project name: Referring to the Eclipse or IntelliJ project;
Repository URLs: The names of the source code repositories related to the project, one project can be stored on several repositories;
Repository version: The source code version in the repository, *i.e.,* commit id of the last checkout/update/pull request;
Test session start: Timestamp (date and time) of the launch of the test runner;
Test session end: Timestamp at the end of the last test execution;
Tests executed: The list of each test executed in the session with the following details:
- **Fully qualified method name:** The name of the test method with its class and package;
- **Test duration:** The duration of the method execution;
- **Test status:** The result of the test: PASS, FAIL (wrong assert), ERROR (unexpected exception), or SKIPPED (*e.g.* annotated with `@Ignore` in JUnit);

The plugins record the tests sessions (if they are launched from within the IDEs) and send the data to a server. The plugins look for either JUnit runs or Maven runs. Tests run out of IDEs are not logged. This can be a concern, primarily for Maven, as it is rarer to run JUnit stand-alone.

1.3 Filtering and Massaging Data

As usual for *in vivo* case studies, filtering and massaging data to get meaningful answers, was a major task. Because of the way we collected data — this in turn dictated by a strong requirement from the company and the developers —, some information was not readily available, *e.g.,* the code changes are not recorded by the plugin but extracted from the source version repository. We discuss here the hypotheses we had to make.

Test session. Gligoric et al. [2014] define a test session as a run of at least one test between two sets of code changes. Beller et al. [2015] split developers work in Eclipse sessions: from the opening of the IDE until its closing. They see Eclipse sessions as natural dividers between work tasks and between work days.

Next, they subdivide sessions in intervals. A JUnitExecution interval is created at the invocation of the JUnit runner (or Maven test build) and ended when another interval starts (*e.g.*, typing interval).

For this experiment, we defined a test session as one execution of the JUnit test runner or of a Maven test job. They can be composed of one or several tests. In Figure 5.1, test sessions are represented by $T1, \ldots T7$.

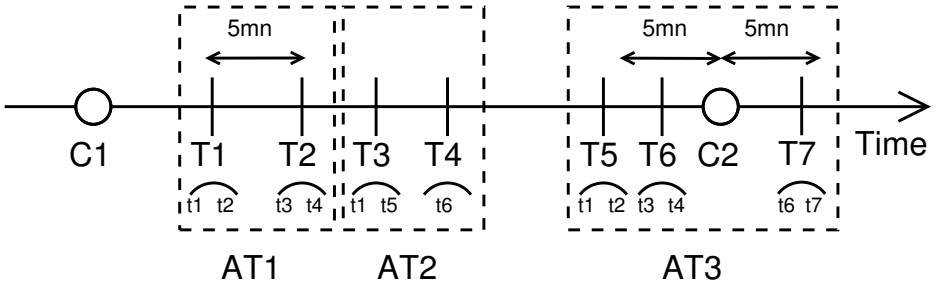


Figure 5.1: A test/code session with three agglomerated test sessions (AT1, AT2, AT3) themselves comprising several test sessions (T1, ..., T7), themselves comprising several tests (t1, ..., t7). C1 and C2 are commits, C1 being the direct ancestor of C2. All events after C1 occur on the same project by the same developer.

Let us consider a developer wanting to run two specific tests from two different classes. With JUnit, the options are either to run all the tests of the project or run independently the two tests. Developers often choose the second option. This means we will have two test sessions. However as far as test selection is concerned, we would like to consider that there was only one "session" including the two tests. However, because we are considering test selection and the tools used are not well suited for it, we have to introduce another concept:

Agglomerated test session. An agglomerated test session is a set of successive tests sessions in the history of a project. Beller et al. also consider test session (called JUnitExecution) and agglomerated test session (called Test Session). Because we do not log every interaction in the IDEs (keyboard or mouse events), we need an heuristic to bound the agglomerated sessions. Two successive test sessions on the same project from the same developer id will be agglomerated as long as:

- The two test sessions occur within a fixed time frame (we chose 5 minutes). Beller et al. [2015] have a similar heuristic with their "reading interval" backed by an *inactivity timeout* of 16 seconds. In Figure 5.1, $AT2$ and $AT3$ are two separate agglomerated test sessions because the time frame between test sessions $T4$ and $T5$ is greater than 5 minutes.
- No single test occurs twice in an agglomerated session. The idea here is that

if a developer runs a test, then changes the code (which we cannot see), then reruns the test to check if it works, we do not want to group both execution of the test as a single group of tests. Gligoric et al. [2014] had the same issue but they can verify whether the developer coded between two executions of the same test or not. This issue is discussed in Session 3. In Figure 5.1, $AT1$ and $AT2$ are separate agglomerated sessions because $T3$ repeats a test also included in $T1$.

Another important issue was to determine what code was being tested. Because we only monitor test sessions and commits (represented as circle in Figure 5.1), it is difficult to know exactly what was the source code tested. For the research questions that require this information (mainly those relating to test selection), we had to use another heuristic and a subset of all the test sessions.

Test/Code session. A Test/Code session is a test session that we could associate with a commit, and thus with the source code that was tested. For this, we group together test sessions and commits that occur on the same project, by the same developer id, and within a time frame of five minutes (similar to the agglomerated test session). This is the case for $AT3$ in Figure 5.1. The five minutes threshold was chosen after manually looking at a number of test sessions and commits that were close. The main difference with “Agglomerated test sessions" is that Test/code sessions are associated to commits whereas “Agglomerated test sessions" were computed independently.

Additionally there may be other test sessions ($AT1$ and $AT2$ in Figure 5.1) between a test/code session ($AT3$) and its ancestor commits ($C1$). Test sessions carry a commit identifier (see Section 1.2). If a Test/Code session ($AT3$) has a ancestor commit ($C1$) whose identifier is associated to test sessions (*e.g.,* $T1$), then, this test session will be added to the Test/Code session. The scenario envisioned here is: the developer does a checkout/update/pull request, changes the code, tests it, makes further changes, tests it more, and finally commits it. In this case, we group the test sessions in the Test/Code session and we assume they all test the code that was committed. This last step is independent of any five minutes interval. This (partial) dismiss of the five minutes threshold is the other difference between “Agglomerated test sessions” and “Test/Code sessions”. In the end, everything after $C1$ in Figure 5.1 is considered one single test/code session.

Code change. We use the test/code sessions to compute code changes. We compare the code in the commits of a test/code session ($C2$, final code) to the code in the direct ancestor of these commits ($C1$, original code).

Amount of code changes. Some research questions require to evaluate the amount of code changed. This will be estimated as the textual (line based) diff between two versions of code. Gligoric et al. [2014] used the number of AST node differences between two versions of the code, but because of the size of the projects and the number of projects, it was intractable for us to use the same solution.

1.4 Automatic Test Selection

To answer research question RQ3.3, we must compare manual test selection done by the developers (corresponding to the tests they launched) with what they should have selected given the changes to the code. For this, we compute code changes (see above) and what tests exercise these parts. We used the Moose static approach solution at a method granularity level as described in the previous chapter. This approach solves all the issues we identified and has the best precision and recall among the ones tested. If one of the methods of the project is affected by a code change, the associated test should be selected and re-launched to check the validity of the change. Our oracle is not perfect but still is a good approximation for a static approach as shown in the previous chapter.

1.5 Interviews with the Participants

To extract more insight from the participants of this experiment, we conducted an interview at the end as suggested by Wohlin et al. [2000]. This study is a mixed analysis as described by Tashakkori and Teddlie [1998]. It is used to add insights on the quantitative data thanks to a qualitative analysis. We followed the guidelines of Hove and Anda [2005] to report the context of the interviews.

1.5.1 Participants

Participants all originate from Worldline. We asked among the employees that installed and used the test recorder plugin if they wanted to be interviewed. The participation to the interviews is on voluntary basis. So, the participants are coming from the 32 developers that previously installed the plugin. We asked them by emails to contact us to be interviewed. However, in front of the low number of answers (only one), we decided to contact all the participants one by one to conduct the interview. On the 32 participants to the previous experiment, 11 participants accepted to take part in the interviews. No monetary compensation was given. These 11 interviews are described in Table 5.1. They are all developing in Java and are using Eclipse or IntelliJ as their IDE. They reported between 4 and 30 years of experience in IT and 4 to 28 in the company.

1.5.2 Study Setting

On face to face in their office, or through video conferences, we conducted 20-30 minutes discussions to realize the interviews. 13 questions have been designed to bring insight for each research question where quantitative data have been already studied. After a brief description of themselves and a quick presentation of their project, we asked them the questions:

Table 5.1: Descriptive Statistics per Participant

	Years of Experience			Job Position
	in IT	in Worldline	on the project	
Alice	9	5	2	Developer
Carol	5	5	3	Architect
Dave	4	4	4	Developer
Eve	9	9	2	Developer
Ivan	30	27	10	Developer
Justin	7	4	2.5	Developer
Plod	20	20	1	Developer
Steve	4	4	2	Developer
Trent	30	25	4	Technical Leader
Walter	28	28	0.5	Technical Leader
Zoe	20	18	0.1	Technical Leader

- Do you consider that your application is sufficiently tested? Why?
- How do you proceed with the tests? Do you test at each change? Before a commit? Along the developments? At the end of the developments? Before releasing to the client? ... Before a change, do you check that the tests are passing?
- What are the difficulties do you encounter to test your application?
- What kind of tests are you launching (Unit, Integration...)?
- Do you launch tests outside the IDE? Why?
- How do you choose the tests to launch?
- How frequently are you faced with a feeling of lack of tests in your application?
- Are you trying to tests all the changes you made on your application?
- For what reasons would you not test a change?
- What is your behavior in front of a failing test? Do you check whether it was green before? Do you correct it? Delete it? Ignore it? Modify it?
- Literature Gligoric et al. [2014] details two strategies to relaunch the tests when some are failing:
  - Launch the tests one by one until it is passing and go to the next one;
  - Launch a same group of tests until they are all succeeding;

  Does your behavior fit itself in one of these groups?
- How do you explain the huge number of green tests versus the low number of tests that fails?
- Do you have the feeling to launch some tests needlessly?

This questionnaire was designed to focus on each of our research question.

It was allowed to answer the questions in any order. Depending on the discussion flow, we preferred to let the respondent answer a question and come back to a miss one after.

The interviewer was the developer of the plugin recording the tests. He has a good knowledge on testing and the Worldline development processes.

1.5.3 Data Collection

To collect the data of the interview, only notes were taken of the answers to the questions. The interviewer was able to write all the answers of the interviewee on paper while conducting the discussion. The discussion flow has not been interrupted.

1.5.4 Data Analysis

For each question, the analysis of the data consists in two steps, the first is to identify the topics that can be extracted from all the interviews. The goal is to obtain a list of topics that answer the question. The second step consists in merging resembling topics and weighting each one with the number of interviewees that share the same point of view. This data extraction was filled into an Excel sheet for a better analysis of the data.

The result of the interviews are integrated in each research question to explain the quantitative results found by monitoring the developers. We compiled the interviews and drew conclusions.

2 Results and Discussion

We now present our results obtained by monitoring the Worldline employees and comparing them to the ones obtained in the literature.

2.1 Case Studies

These results were obtained between April 20th, 2016 and March 8th, 2017. Tables 5.2 and 5.3 present some descriptive statistics on the case studies.

We have 32 participants in 64 different projects which sets us in between both other case studies, closer to the paper of Beller et al. [2015].

We have more test sessions (14 686) than Beller et al.'s paper (3 424) with fewer participants (see also session/developer). We also have an order of magnitude more single test executions compared to Beller et al.'s paper (153 763 for us; 10 840 for them). This can already be seen as a good indication for test practices in the

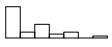
Table 5.2: Descriptive statistics on the three case studies

	Beller et al. [2015]	Gligoric et al. [2014]	Worldline case study
# Developers Executing Tests	48	14	32
# Projects	73	17	64
# Test sessions	3 424	5 757	14 686
# Agglomerated sessions	–	–	13 611
# Test executions	10 840	264 562	153 763
# Unique Tests	–	6 560	15 249
Tests / Session	3.2	45.9	10.5
Sessions / Developer	71.3	411.2	458.9
Study Duration (months)	4	3	10

company. But comparisons are made difficult by the fact that we do not know what development principles are used in each environment, *i.e.,* open-source, student, and industrial.

We are also intermediary in the number of single test executions per test session (10.5) and above all for the number of sessions per developer (458.9). This number of test sessions per developer is higher than the open and closed source projects which is a good sign for the company.

Table 5.3: Descriptive statistics per developer

	Min	Q_1	Median	Q_3	Max	Histogram
Calendar Days	1	60	193	241	326	
Activity Days	1	6	23	54	140	
Sessions	3	32	192	706	2343	
Sess./Activity Day	1	3	8	18	110	

As an additional indication, we give in Table 5.3 statistics for developers of our case study: number of days of collecting data, number of test sessions, and number of sessions per day where tests have been made (activity days), all developers combined. We note that developers participated in the case study for more than 5 months on average and did 61.7 testing sessions per activity day. From this last number, it seems that testing is well implanted in company's developers daily practice. However, by running the interviews, it seems that in a majority of projects of the company, developers do not run tests automatically during the development but

only manually or at the integration testing step. There is no figure on the number of developers running automatic tests in the company: only developers running tests installed the plugin.

2.2 RQ1: How and why developers run tests?

2.2.1 RQ1.1 Do developers test their code changes?

For this question, we evaluate whether there is a correlation between the number of test runs and the number of changes to source code. We used Spearman correlation as our data do not follow a normal distribution. For Worldline, the correlation is weak $\rho = 0.20$ confirming that more code changes do not lead to more tests.

Beller et al. [2015] differentiate the number of changes to test code from number of changes to production code. They correlated them both to the number of test runs. They have a good correlation ($\rho = 0.66$) with test code changes, and a weak one ($\rho = 0.38$) for production code.

From the interviews, nine developers want to test all the changes they made in the application. However, some pieces of software like human machine interfaces, insertion databases, dataset creation, or complex systems require specific testing frameworks. These frameworks are difficult to put in place due to a lack of training and time to understand it. Moreover, six Worldline developers confess that they do not run tests after minor changes, and six because of a lack of time.

2.2.2 RQ1.2: How long does a test run take?

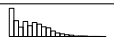
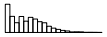
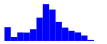
We observe that 50% (median) of our test sessions finish in less than 3 sec. and over 75% (third quartile) of tests sessions finish within 16 seconds (results are similar for agglomerated sessions with respectively 3 and 18 sec.). Moreover, 9.4% of the test sessions take longer than 1 minute and 5.0% take longer than two minutes (respectively 10% and 5.6% for the agglomerated sessions). Detailed results can be found in Table 5.4. In general, tests sessions are short.

We measured a maximum duration of the test sessions of 4 h 23. In this session, only one test was launched. Other executions of this test take few seconds to run. But, the results being anonymous, it is not possible to ask the developer for more information on this long duration.

Beller et al. report that 50% of their test sessions finish in less than 0.5 seconds and over 75% of the sessions finish within 5 seconds. For their test sessions, 7.4% take longer than one minute and 4.8% take more than two minutes. They conclude that most of the test sessions are short.

Results are comparable, orders of magnitude are the same excepted the duration of the test sessions. On this point, one could hypothesize that tests are broader

Table 5.4: Comparison of our results with those of the Worldline Case Study. (When computing number of tests per session, we give results for test sessions and agglomerated sessions to match Beller et al.'s case study). Histograms are in log scale

			min	Q_1	median	Q_3	max	*unit*	Histogram
RQ1.2	Test session duration	Worldline (test sess.)	0	1.0	3.0	16.0	15 820.0	*second*	
		Worldline (agglom.)	0	1.0	3.0	18.0	15 820.0	*second*	
		Beller	0	0.03	0.5	3.4	73.8	*second*	
RQ1.4	Percentage of executed tests	Worldline	0	1.2	4	17.8	100	%	
		Beller	0	1	1	12.5	100	%	
RQ2.2	Time to fix failing tests	Worldline	0	3.1	4 981.0	1 042.0	359 600	*minute*	
		Beller	0	1.7	65.1	25.0	4 881	*minute*	

in scope in our company. We confirmed through the interviews that seven participants are executing more customer tests than developer tests. Worldline customer tests require to set up a database and have a higher number of method tested than developer tests. Five developers said that the tests take time to be executed.

2.2.3 RQ1.3: Do quick tests lead to more test executions?

To answer this question, we evaluate the correlation between test execution length and the number of times tests are executed. The expectation is that short tests will be executed more often, thus the correlation value is expected to be negative. However, in Worldline study, Spearman correlation value was $\rho = 0.20$. Beller et al. get a positive correlation value of $\rho = 0.26$. Both lead to the conclusion that there is no correlation.

These answers are contrary to expectation, faster tests are not executed more often (corollary: longer tests are not executed less often). This was one of our hypotheses to try to improve test practice in the company and it does not hold.

Interviewed peoples seem to launch the tests that cover the part of the application they changed without distinction of the duration of the test. One of the interviewees said that long tests were not frequently executed. As an additional verification, we decided to apply another statistical test. We grouped the tests by their duration: tests of less than 10 sec.; tests between 10 & 20 sec.; ...(see Table 5.5). Then, we verified whether the median number of executions of the groups were different with a Wilcoxon statistical test. The only statistically valid difference is between group 1 and group 2. But, it must be noted that group 1 contains many more data than the other groups. This could affect the results.. We could not show any significant difference between any of the other group. Consequently, we cannot accept on our data that long tests are executed less often.

Table 5.5: Test Duration and the number of execution of each test

	<10"	10"- 20"	20"-30"	30"-1'	> 1'
# Tests	1404	243	84	65	78
Median # of executions	5	8	8	12	7

2.2.4 RQ1.4: Do developers practice test selection?

For Worldline, we report 58% of the agglomerated sessions with only one test, 24.5% with more than 5 tests, and 4.0% with more than 50 tests. We can reach the conclusion that developers of the company practice test selection.

Beller et al. report that 87% of test sessions include only one test case, 6.2% include more than 5 tests, and 2.9% more than 50 tests. From this, they concluded that their developers did practice test selection. Gligoric et al. report 3 594 test sessions (62.4%) with only one test.

It seems Worldline's developers and those in the second paper's case study Gligoric et al. [2014] select less "aggressively", *i.e.,* with fewer test sessions consisting of only one test.

For Worldline, in 50% of the test sessions, 4% of the available tests of the project are selected, in 75%, 17.8% are selected (See Table 5.4).

Beller et al. further note that in 50% of the test sessions, only 1% of the available tests of the project are selected, and in 75% of the cases, 12.5% are selected.

For us, almost all the tests (> 95% of all the tests) are selected in 1.8% of the test sessions, and, for Beller et al., all tests are launched in 3.7% of the cases

So developers of the company select more tests than those of the first paper when they select, but they execute all the tests available much less often, almost always doing test selection.

We report that test selection occurs in 81.4% of the studied test sessions, between Gligoric et al. (59.19%) and Beller et al. (about 96.3%[1]). Finally we report an average selection ratio (number of executed tests divided by number of available tests) of 8.8%. For Gligoric et al. this ratio is almost the same with 9.0%. So again, it seems that Worldline developers tend to select more tests when they select.

Thanks to the interviews, we identified several profiles of testers (a developer can have several profiles): Six developers run all the tests of the module or subproject where the modification has been made. It is the preferred solution if the tests are fast, else, developers select more rigorously the tests. Six developers run tests based on naming conventions: test class has the same name than the application class. According to the interviews, in another group gathering 7 developers, they select tests according to their feelings and experience. The testers feel they know what tests are potentially affected by the latest changes. Finally, we found three developers using the call graph available in the IDE to retrieve the tests to relaunch. It is an advanced approach to select the tests.

2.2.5 RQ1.5: What are common scenarios for manual RTS?

Gligoric et al. [2014] identified two common patterns for test selection:

- "After one or more tests fail, developers usually start making code changes to fix those failing tests and keep re-running only those failing tests until they pass. After all the failing tests pass, the developers then run most of, or all the available tests to check for regressions."

[1] Our statistics from their numbers

- "[Developers] fix tests one after another, re-running only a single failing test until it passes."

By analyzing the data, we found these two patterns in our company. In the interviews, developers said that they launch tests one-by-one to avoid side effects between the tests. They also run semi-automatic tests one-by-one: they run the tests injecting data in database automatically and check manually the result. But both scenarios are equally frequent and depend of the current step of the workflow the developer is into. Launching groups of failing tests is made when the tests are jointly failing and cover the same feature(s) of the application. One-by-one launch is frequently used when only one feature needs to be checked and there is only one test associated to it.

2.3 RQ2: How do developers react to test runs?

2.3.1 RQ2.1: How frequently tests pass and fail?

For Worldline, on 153 763 tests executions, the ratio of failing tests is 13% (20 272), and the ratio of passing tests is 83% (127 704). We can also report 4% (5 787) of skipped tests. In Beller et al. [2015], on 10 840 tests executions, 65% (7 047) fail and 35% pass successfully.

We found a much lower ratio of failing tests in our case study. By interviewing Worldline developers, we can propose some explanations:

- The tests are launched and followed up. This shows better testing practice in the company that does appear in Beller et al. [2015].
- The tests are passing because they miss assertions to check the behavior of the application. So in reality, the test should fail.
- The tests are not really tests but are launching scripts to insert fields to set up the database.

But, ten developers of Worldline said that they fix tests when they are failing. This also explains the high number of green tests.

2.3.2 RQ2.2: How long does it take to fix a failing test?

In the Worldline case study, around 12% (1 780) of the tests are never fixed. For the failing tests that get fixed, 50% are resolved in approximatively 20 minutes and 75% within approximately 17 hours 20 minutes. The maximum duration that we observed to fix a test is 249 days, 17 hours and 20 minutes, almost the duration of the entire case study (322 days).

In Beller et al. [2015], for 70% of the tests (2 051), the authors observed at least one successful execution and 30% have no successful execution. Therefore a significant part of the tests are never fixed. For the 2 051 failing tests that are fixed

at some point, 50% are executed again with success within 10 minutes and 75% within 25 minutes.

Results for this question can also be found in Table 5.4.

Our longer delays could be caused by the fact that the tests in the company are broader in scope. They mainly implies complex environment with database or external applications. As already discussed in RQ1.2, broader tests would make it more difficult to pinpoint the error when they fail.

2.4 RQ3: How and why developers perform test selection?

2.4.1 RQ3.1: Does manual test selection depend on size of test suites?

On all Worldline projects we studied, all developers performed test selection. On the other hand, we have an average of 254.1 tests per project with a minimum of 1 and a maximum of 2 216. We can conclude that developers performed manual test selection regardless of the size of their test suites. In Gligoric et al. [2014], almost all developers performed manual test selection, and they also had a wide range of test suite sizes. They further report an average of 174.3 tests per project; the minimum was 6 tests, and the maximum was 1 663 tests. The authors finally add that "considering that these projects are of small to medium size, and because they exhibit manual [test selection], [they] expected that developers of larger projects would perform even more manual [test selection]."

Through our interviews, it appears that the test selection depends of the number of changes they made. The projects are frequently split into modules and each module contains its own tests. So, if a change occurs in one of the modules, the developer relaunches all the tests of the module. If the developer knows which test is related to the part he changed, he selects only few tests to be relaunched.

2.4.2 RQ3.2: Does manual test selection depend on size of code changes?

We consider the relationship between the size of recent code changes and the number of tests that developers select in each test session. For Worldline, the correlation value is $\rho = 0.11$, so we can conclude that there is no correlation. For Gligoric et al. [2014], their correlation value is $\rho = 0.28$ which also means that there is no correlation.

The conclusion from this research question is that one would expect developers to run more tests after large code changes or to perform more test selection when there are more tests in a project. The findings go against both assumptions. We may relate this to RQ1.3 where we noted that faster tests did not lead to more execution and to RQ3.1 where developers avoid to select tests that takes more than one minute to run. There seems to be convergence of evidences that, contrary to

our hypotheses, test selection and execution are not significantly influenced by the duration of the tests or their number.

Interviews conclude that developers are potentially running more tests if the changes they made are in several modules: they run the tests of all modules impacted. But, most of the time, changes are located in only one module.

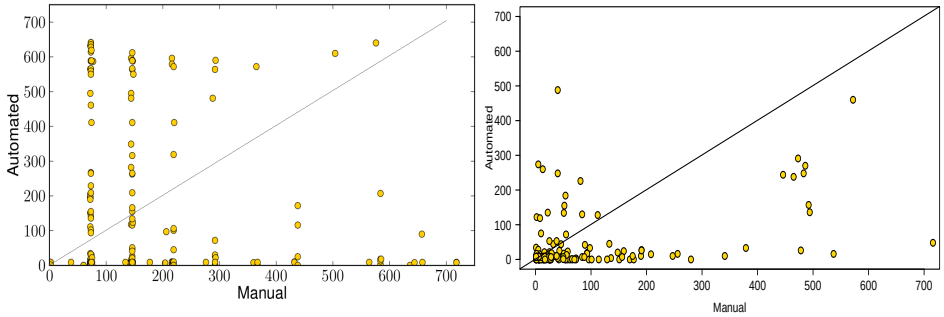


Figure 5.2: Relation between the number of automatic and manual test selection (left Gligoric et al. [2014], right our case study)

2.4.3 RQ3.3: How does manual test selection compare with automated one, in terms of precision and safety?

We present the comparison of manual versus automated test selection in a dot plot (Figure 5.2, right). Numbers of expected tests selected (automatic test selection) are represented on the Y axis, and number of actually selected tests (manual test selection), on the X axis. The desired behavior would be to have all points on the diagonal $x = y$. Points above the diagonal indicate that the manual selection missed some tests (low recall, assuming the tests selected are all correct). Points below the diagonal indicate that the manual selection chose undesirable tests (low precision, assuming no needed tests were missed). Our study reports a correlation of $\rho = 0.16$ between the two metrics which confirms the visual impression of no correlation.

Gligoric et al. [2014] whose data are on the left side of Figure 5.2, conclude also to the absence of correlation with a value of $\rho = 0.18$.

Our case study can further report a precision of 37.43% (ratio of selected tests that are correct) and a recall of 28.77% (ratio of required tests that were selected) which should be considered low results. The conclusion is that manual test selection is not accurate, which was expected.

From the interviews, developers said that they do not always carefully select tests: sometimes they are launching more tests than required (they relaunch all the tests to test the whole application), or not enough tests (to test only the algorithm

they just implemented). When the modification in the source code is minor or deals with a graphical part of the application, developers do not always run the associated tests. All of the interviewees said that are not launching useless tests.

3 Threats to Validity

This section discusses the validity of our case study using validation scheme defined by Runeson and Höst [2009]. The construct validity, the internal validity, and the external validity are presented.

3.1 Construct Validity

Construct validity indicates whether the studied measures really represent what is investigated according to the research questions. The purpose of this study is to evaluate the behavior of developers of the company about testing.

We detected that developers may use "false" tests as a standalone application to run a server and make manual tests. We manually removed these by looking at all test sessions longer than 10 minutes and having only one or two tests. It could be the case that such tests are still present in our data for example if they were launched as part of a session with three tests but removing them all would require manually analyzing every test (little less than 7 000).

The plugin records only the execution of JUnit and Maven test sessions if they are launched from within the IDEs. If other tests runners are launched from outside the IDE (less probable for JUnit than for Maven), we would have no trace of that. This is a common issue that Beller et al. [2015], Gligoric et al. [2014] also had. However, the question was asked in the interviews. It appears that most of the automated tests are launched by the developers in their IDEs. Only a couple of developers are running tests through a maven command line. But it is mostly to deploy the application or because the tests are too long.

The presence of continuous integration is another bias that can persist in the case study. Despite the fact that developers should launch the tests locally to avoid committing potential bugs and propagate them to their colleagues, they tend to delegate this validation to the continuous integration. Consequently, fewer tests are made locally and potential bugs are dispatched to the others developers of the team.

Associating tested code and test sessions is still a real issue. The ideal solution would be to record all the source code for each test session. But this would mean a much more intrusive plugin that we are reluctant to install on the developers computers for now. One action that we may take would be to check whether our current (imperfect) solution makes a noticeable difference in the result compared to

a "perfect" one. This can actually only impact the results on the computed precision of the test selection.

3.2 Internal Validity

Internal validity indicates whether no other variables except the studied one impacted the result.

The developers know that they are under study, Hawthorne effect[2] may have taken place [Mayo, 1933].

The sample may be biased towards developers who are actively interested in testing because participation was voluntary. In this sense, our results could be an overestimation of the real testing practices.

Between the start of the study to its end, six participants left the company. To encourage participation of less testing aware participants, we organized a lottery (Beller et al. did the same), but it only brought four additional participants.

3.3 External Validity

External validity indicates whether it is possible to generalize the findings of the study.

Participants to the case study all originated from the same IT company or from the same language community. This would point toward a real threat to generalization. However, the fact that we make our case study close to the Beller et al. [2015], Gligoric et al. [2014] ones, in different conditions, and mostly confirmed their results, seems to be a good enough guarantee.

As Beller et al. [2015], Gligoric et al. [2014], all projects we studied are in Java (as 80% of the projects developed in the company). This can be an issue to generalize on other programming languages where testing is more difficult.

Moreover, testing practices of 32 Worldline employees during more than 10 months were studied. It is a case study of the partial state of testing in a major IT company and can be difficult to generalize to other companies.

4 Conclusion

A first step before providing the Worldline developers with a test selection tool is to know their behavior about testing. So, this empirical study was inspired from two previous case studies [Beller et al., 2015, Gligoric et al., 2014]. Our study took place in an industrial, closed source context, whether Gligoric et al. [2014] case study was conducted with students (mainly), and Beller et al. [2015] one was

[2]Tendency of people to work harder and perform better when they participate in an experiment.

conducted on various projects including open-source and students projects in the other case [Beller et al., 2015]. Thus it was not clear whether their conclusions applied to industrial closed source environments. We could confirm many findings of both papers, thus giving more weight to their conclusions. Our conclusions are:

- Test practice in the company is better compared to practice described in Beller et al. [2015] with open-source, students and industrials. It was unexpected but could be biased by the voluntary participation to the experiment;
- Developers do perform test selection, mostly reducing the test suite to one test (more than half of the test sessions ran only one test). This is coherent with previous findings;
- Manual test selection is not accurate, many impacted tests are not launched after a change (recall=29%) and others are whereas they did not need to (precision=37%). We noted a tendency to err on the side of minimality rather than safeness. This is in contrast with the reports from Gligoric et al. [2014] (73% of the sessions executed more tests than an automated RTS would have);
- Contrary to intuition, amount of test execution or test selection do not depend on the size of the test suite nor on the duration of the tests: Shorter tests are not executed more often, and the number of test runs is not impacted by the total number of tests available in a project. Interviews confirmed that tests are actually selected on their ability to confirm the quality of a code change. This is good news at it reinforces the need to provide adequate test selection mechanisms to help software developers getting faster and better feedback.

Now, with the testing approach adapted to the Worldline developers, and a better knowledge of their current testing behavior, we are able to propose them a tooling allowing them to be more efficient in their everyday work.

CHAPTER 6

Impact of the Usage of the Test Selection Tool

Contents

In previous chapters, we saw that there is no tooling adapted to the Worldline environment to perform test selection. In our goal to try to improve testing behavior, we developed a tool implementing the features we found missing. The goal of this tool is to change the developers' development process. Chapter 2 showed that one of the causes of project failure is the bypassing of tests. We hope with our tool that developers will launch more tests. These tests should be compatible with the changes the developers made in their source code. To know if such a tool can achieve this goal, we evaluated its usage through data analysis and interviews of the Worldline developers that participated in the study.

1 Test Selection Plugin

A test selection approach selects the tests that exercise the changes the developer made. The approach should provide fast feedback to be used by the developers in their daily tasks. So, as soon as the developer writes code, the tool has to be able to select the possibly impacted tests and launch them.

Developers are writing code inside their IDE. Swapping the application to get the results of test selection would make the development process more complex. On the other hand, the IDE integrates the test runner and project configuration which are already configured by the developer. An external tool cannot access this information whereas a test selection plugin integrated in the IDE can. We consequently developed such an IDE plugin. In a first part, we describe the main features of it and, in a second part, we detail the implementation.

1.1 General Overview

1.1.1 Purpose

The purpose of the test selection tool is to automatically select the tests related to the source code developers changed. We expect that the tool will encourage them to test their applications more often. Furthermore, their test sessions should become more complete and more frequent.

But, to encourage developers to use the plugin, it should be adapted to the constraints of Worldline's developers whose time is scarce. So, the time to learn how to use the tool should be reduced to a minimum: the plugin has to be simple to install and to use. Developers should be able to install it like any other plugin. It should not cause any slow down in the development process. Such a slow down could make the developers stop using the plugin or uninstall it. Developers use both Eclipse and IntelliJ. Thus, we developed a plugin for each IDE.

1.1.2 Test Selection Approach

In Chapter 5, we saw that developers tends to err on the side of minimality instead of safeness in their test selection. That is to say, they prefer to run less tests than all the tests that might be affected by the changes they made. We also saw that developers often reduce their test suite to only one test. Their test selection is not accurate and many possibly impacted tests are not launched after a modification (only 29% of the tests that should be launched, are really launched). So developers miss opportunities to run tests that may potentially fail. Such tests failures will be detected only after some time, for example, the following day as a result of the integration, and developer will loose time to find its root cause.

Between the approaches to select tests at our disposal (see Chapter 4), the static approach navigates through all the possible paths from the changed source code parts to the impacted tests. As explained, it will select more tests than a dynamic approach.

On the contrary, a dynamic approach is more precise and will select less tests. But it requires to relaunch all the tests at the beginning of the development session, or to update regularly the mapping between tests and source code after changes. As already said, executing all the tests may take hours on some projects of the company. Such a wait is not acceptable. Consequently, we choose to use a static approach rather than a dynamic one.

In order to be more accurate in the test selection and not spread useless information to the developers, we decided to integrate in our plugin an approach at a method granularity. So, only the exercised tests methods are selected by such an approach and not the other tests methods of the exercised classes as it would be in the class granularity approach.

1.1.3 Tool Operation

The workflow that we want to provide is the following: The tool listens to file changes in the IDE. As soon as the user saves a source code file in the IDE, test selection is triggered, tests are executed, and test results are listed in the interface. This workflow is the default one and other strategies are defined to change it (these strategies are defined later). Figure 6.1 describes this process: an event, depending on the selected "Test Selection Strategy", triggers the test selection, then, depending on the "Test Execution Strategy", the tests selected are launched or not, and test execution results are displayed in the interface.

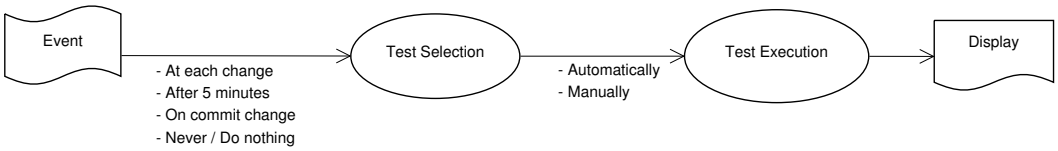


Figure 6.1: Test Selection Tool Workflow

Some "Test Selection Strategies" are implemented to trigger the selection of test depending on the action of the user:

At each change. This is the default strategy for test selection. The exercised tests are selected immediately after a change in the source code (save, file update,...).

After 5 minutes. All changes are recorded. Every five minutes, test selection is triggered on all the recorded changes. The five minutes threshold was chosen arbitrarily as a reasonnable delay to allow for some work to be done while not having too much to test at a time.

On commit change. Test selection is triggered when the developer commits inside the IDE. Changes are considered since the last test selection.

Never / Do nothing. Test selection is not triggered, but changes are recorded. As soon as another strategy is selected, these changes are considered for next test selection.

The "Test Execution Strategy" allows the developer to run the tests as soon as the test selection has been computed. There are only two choices: the test execution is launched automatically, or, a list of tests to run is created: the user can then execute the tests of the list manually when he is ready.

1.1.4 User Interface

An example of the display of the tests execution for Eclipse is shown in Figure 6.2.

The "Test Execution Strategy" appears as a checkbox labeled "Auto run selected tests" which is checked by default. If the developer decides to disable automatically execution, tests are still listed and a button allows to launch them on demand.

The list contains the name of the launched tests, their launching timestamp, their Eclipse project and the color corresponding to the test result: Green stands for test passing, yellow for assertion failure, and red for error. The color is gray while tests are not launched. This interface also allows the user to select a different strategy for the test selection (combo box on the left).

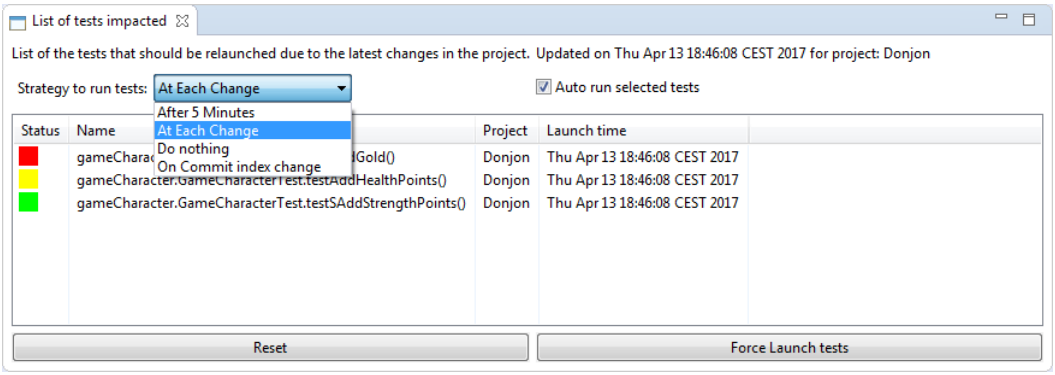


Figure 6.2: Display Window for Selected Tests in Eclipse

1.2 Architecture

1.2.1 Client Server application

The test selection plugin has to be written in Java. But, we would like to reuse the test selection algorithms we implemented in Moose (see Chapter 4). So, we created a REST API as a bridge between both to exchange information.

Moreover, Moose is a standalone application that takes almost 100MB of disk storage. Shipping this application to the computer of the developer would have been too invasive. Consequently, Moose was installed on an internal server of the company, where each developer plugin requests the list of tests to select.

1.2.2 Workflow

Figure 6.3 illustrates the workflow of the test selection tool. On the client side of the plugin, a model of the source code is created before any change occurs in the

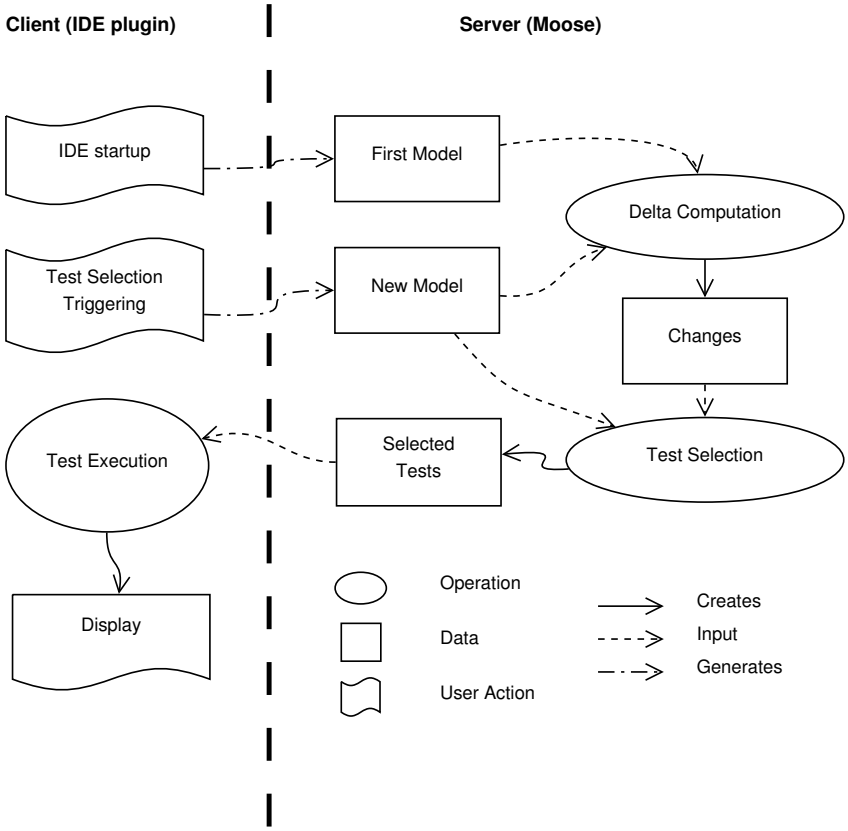


Figure 6.3: Test Selection Tool Workflow

environment of the developer. This First Model is created at the IDE startup to ensure that no change is missing. It is then sent to the server as baseline.

Then, the developer makes changes in his source code. Only the changes done inside the IDE can be logged by the plugin. Following the selection strategy (see Section 1.1.3), the plugin triggers a new parsing of the source code and generates a new model. This new model is sent to the server and loaded into the Moose application. It represents the model of source code changed by the developer.

The server now contains two versions of the source code, one without the changes of the developer, and another with. The changes between the two models are computed to finely identify the modifications that occurred on the source code. Famix Diff is a tool based on Moose to compute changes between models. It is based on the algorithm described by Xing and Stroulia [2005] to detect the structural changes between two versions of a modelised object-oriented software.

Then, the test selection approach is made following the Moose approach at a method granularity which takes into account all the problems we defined in Chapter 4. This test selection is performed by navigating the new model from the changes computed by Famix Diff

The list of selected tests is sent back to the IDE plugin and displayed to the user. Depending on the execution strategy selected, the tests are executed or not. Since the majority of Worldline's developers use JUnit to write the tests, we execute them with the JUnit runner integrated in the IDE. After the execution of the tests, the display is updated to show the actual results of the tests.

A new cycle can start again with the last model as basis.

2 Case Study

The plugin described in the previous section was announced by email, few developers installed it. We analyzed the change in their development processes by analyzing data and interviewing the developers.

2.1 Data Analysis

To the 32 developers from the initial experiment, we sent emails and direct messages. Once every developer answered, only six accepted to install the plugin. This low number can be explained by some turnover in the company. One of the 6 participants who installed the plugin, launched a few tests and dropped the experiment the day he installed it. He did not see an immediate interest in it. We discarded his data in the experiment. Another one was on a mission on a project where he has to improve the testing process of the project team. It seemed a perfect candidate for us. However, we encountered some problems due to very large number of lines of

code of the project. When the plugin was fixed after couple of weeks, his mission was finished. He then went to a new project where he is not launching automatic tests anymore. Consequently, we have data for only four developers.

To protect their anonymity, we will named them Alice, Bob, Charlie, and David. Some descriptive data on their background can be found in Table 6.1.

Table 6.1: Descriptive Data of the Participants

	Job position	Years of Experience	Team size
Alice	Senior Developer	5	6
Bob	Senior Developer	3	3
Charlie	Junior Developer	3	6
David	Junior Developer	2	12

Table 6.2 depicts for each user the following data:

Number of days. The number of days between the first day the developer launches a test and the last one.

Number of manual test executions. The number of tests executed manually, *i.e.,* by clicking on a button.

Number of automatic test executions. The number of tests executed automatically, *i.e.,* selected by the plugin and launched automatically.

Total number of executed tests. The total number of tests executed.

Activity days. The numbers of days the developer launch at least one test (automatic or manual).

Table 6.2: Testing Behavior Description for each Participant

	# days	# activity days	# manual test executions	# automatic test executions
Alice	89	11	78 (57%)	59 (43%)
Bob	17	4	40 (49%)	41 (51%)
Charlie	35	6	83 (92%)	7 (8%)
David	14	3	9 (100%)	0 (0%)

All the participants kept the default test selection strategy which is to launch the test selection at each modification they made in their source code. In this study, we acquired too few data to study it from a statistical point of view. The number of participants and the number of test executions are too low: only 4 developers took

part in the study. We consequently preferred to analyze the participants' behavior independently one from the other and reinforce this study by interviewing them.

In the following, we describe how the qualitative study with developers' interviews was planned and we detail the individual results for each developer.

2.2 Interviews Description

The study consisted in semi-structured interviews with the participants who installed our test selection tool. All of them (four) accepted to answer our questions. These interviews only aimed to explain the few quantitative results we had. Two types of questions were defined: one part is general for all the interviewees, the other is specific for each participant based on the data we collected and the question they raised.

2.2.1 Global Questions

The general questions we asked to all the participants are the following:

i. What do you think of the relevance of the selected tests? More specifically, four possibilities exist:
   - Selected tests are correct but some others are missing (good precision, bad recall)
   - All impacted tester are present but there are too much (bad precision, good recall)
   - Some test are missing and some are wrongly selected (bad precision, bad recall)
   - Yes, there are exactly the tests that should be selected (good precision, good recall)

ii. The tool suggests several strategies for the selection and execution of the tests. Did you use them all? Would another test selection or execution strategy like to suggest?

iii. Did your development process change?
   - Do you launch more or less test than before?
   - Do you commit more or less often?
   - Do you write more or less tests? Do you modify the existing tests to insert new assertions?

iv. Do you find it easier to fix the tests? Is the debugging of the test faster?

v. Do you trust more your committed source code? Or the code committed by the others?

vi. Do you have others comments, questions?

2.2.2 Individual Questions

For each participant, we did a specific analysis. We consequently added some questions on some specific points to clarify.

Alice

Alice is one of the first users who installed the test selection plugin. At the same time, she executes tests both manually and thanks to the automatic execution of the plugin.

We discovered that the test selection algorithm sometimes selects a skipped test that she never launches manually. A skipped test is annotated with `@Ignore` by developers who do not want to launch them. It is often tests that are deprecated due to recent changes in the application. Despite this test covers the changes in her application, Alice chooses to ignore this test and not enable it. We added a specific question:

- Why are you skipping several times a test that is selected by the tool?

Bob

Despite the fact that Bob used the tool during 17 days and manually launched some tests, all the tests executed returned an error status. The goal of test execution being to ensure that the application is error free, the presence of only failing tests cannot give any confirmation on its quality. Consequently, we expect that some automated tests he launches are passing. Moreover, he launches other tests than the one suggested by the tool. The questions we would ask to him specifically are:

- You seem to launch complementary tests that are not selected by the tool. Why are you launching them?
- All the tests you launched with the tool installed are failing. Why?
- What did you expect from automatic testing then?

Charlie

Charlie deactivated automatic execution of the tests. Only seven tests out of 90 he launches are executed automatically. We can hypothesize that the tests selected are not accurate or are too intrusive for his usage. We asked him these specific questions:

- Why did you deactivate the automatic execution of the tests the tool suggests after the changes you made?
- Do you think that test selection was not reliable?

David

David also deactivated the automatic launching of the tests after their selection by the tool. He used the tool on only three days in a 14 days period and uninstalled it. The questions we ask to him are:

- Why did you deactivate the automatic execution of the tests the tool suggests after the changes you made?
- Do you think that test selection was not reliable?

Three of the interviews took place through audio-conferences. They lasted 10 to 15 minutes.. The last one was a face to face meeting. All four participants to the experiment agreed to be interviewed.

3 Results and Discussion

We first give some global results on the interviews, and then, we detail for each participant the specific questions we asked them.

3.1 Global Results

On the four developers, three declared they used the tool only a few days: Bob and David found that it slowed their development process due to a high usage of computing resources on their laptop. We did not establish this issue during our first trials: the projects of the company are diversified and we did not tested on large enough projects. Charlie used it but as he is leaving the company soon, he will not continue to write tests after the experiment. The fourth, Alice, used a lot more the tool and still uses it.

3.1.1 What do you think of the relevance of the selected tests?

Three developers found that the test selection had a high recall and a high precision, *i.e.,* all the tests that are selected correspond to the changes made and there is no missing test that should be relaunched. Charlie thinks that test selection is "pretty cool" and Bob finds that it is "coherent" which the selection he would have made. However, for David, there is too much tests selected. Some tests are not related to the latest changes he made. We are not able to say if it is an issue in the approach or if the developer is mistaken: we do not monitor the changes that lead to the selected tests. One bias of the static solution is indeed to select more tests. Call graph navigation can give results that are outside of the knowledge of the developer. He will not see a direct relation between the source code and the test, but there is one.

3.1.2 Did your development process change?

For the four participants, the development process did not change. They are not committing more or less. They do not modify the tests by adding new assertions to ensure that their code is well covered. Three of them said that they are not writing more tests due to tight project schedule. The other one, Charlie, tends to write more tests. The plugin encouraged him to keep focused on the tests during the development and keep them up to date. For Alice, the tool gives her a visual help on the modifications of her source code.

Only the developer that really used the plugin found a real advantage to his current development tasks. Maybe the study is too short to observe a change in the developers process.

3.1.3 Do you find it easier to fix the tests? Is the debugging of the test faster?

Our plugin enables an immediate feedback on the code changed if it is tested. Consequently, if tests fail, the developer still has in mind the change he just made. He does not need time to remind the changes he made and why.

For two developers, it is not easier to fix the tests. It just helps to select the tests to relaunch. Alice and Bob found a real advantage to the tool. The two developers take less time to identify the failing tests and to modify the related source code. They anticipate potential future failures and are more reactive. It is possible that this fast feedback on the tests allows the developers to not loose time reading again their source code and remembering which part they modified.

3.1.4 Do you trust more your committed source code? Or the code committed by the others?

Alice, Bob, and Charlie, said they trust more the code they committed. It is not the case for David. For the first three, they find it reassuring that tests are passing before committing. Bob said that he knows it is a best code practice he shall apply in his developments.

If other developers of their team also used the plugin, Alice, Bob, and Charlie would trust more the code. For Alice, this trust is also assured by setting up a continuous integration process launching tests periodically. Our tool adds confidence because continuous integration is only launched during the night: modifications on the source code of the other developers are not checked during the day. Developers may share bugs inside the team. Moreover, it is possible that our test selection plugin decreases the number of failing tests on the continuous integration.

3.2 Individual Results

Alice

Alice said she deactivated the automatic test execution. She prefers to choose manually when the tests should run. It is too heavy in the process to run it at each file save. Features are often implemented on several files, so, launching the test selection in between is not interesting.

Alice skipped tests because they call external programs that are not mocked. The mocking is used to fake the answers of method results that are not implemented but should be used to make the application work. More time is required to modify the tests to make them passing again. They do not have this time yet on her project.

Bob

Bob did not change the test execution or selection strategies in the test selection tool. However, he said that a problem in the source code cannot be revealed if the tests are launched too early. A feature in the source code often needs to be implemented in several classes. If the tests are launched as soon as the first class is modified, they fail because changes in other classes are mandatory to make them pass. As Alice, Bob made modifications on several files before trying to compile again and launch the tests. Current strategies of the tools are not adapted to this behavior. Several file saves should be taken into account before triggering test selection. New strategies could be proposed in the tool to avoid this frustration.

For Bob, all the tests selected by the tool ended in error because JUnit runs are not configured in his environment. Actually, he uses Maven to launch the tests instead of JUnit. But Maven does not allow to perform fine test selection without using advanced features which are complex to set up. Consequently, the tool provided him a list of tests to watch but he was not able to launch them.

Charlie

As Alice and Bob, Charlie found that the triggering of the test selection is too frequent when he modifies several files. He prefers to launch manually the test execution when he finishes to implement some features. Moreover, some automatic tests are integration tests that are too heavy to be launched often. He suggested to set up a filter in the tool to remove these long tests from the selection.

David

David did not know about the strategy to select the tests, so he used the one by default at each save of a file. He deactivated the automatic execution of the tests and launch them manually when he finished the implementation of a feature. He said that there are too much tests impacted by his modifications and he prefers to

launch them himself. That way, his development process is not disturbed by the tool.

However, his test selection is more efficient with the tool. He also finds that he should write and launch more tests. According to Hurdugaci and Zaidman [2012], a tool that emphasizes the tests covering a changed method is more adapted to help developers to test their applications.

4 Conclusion

To try to improve testing behavior of Worldline developers, we developed a new tool allowing them to automatically select tests on their projects after each code change. Few Worldline developers installed the plugin. Considering the lack of data, we decided to perform interviews with the four developers which installed the test selection plugin.

They have mixed opinions about the test selection tool: On the one hand, they spent less time to find the tests they should relaunch after a modification in their source code. But, on the other hand, the test selection tool slows the developers in their daily tasks when tests executions are performed at unappropriated moments. as we expected. Finally, the developers suggested some improvements:

- Add some test selection strategies,
- Improve the computing performance of the tests to relaunch.

But, they are all ready to continue to use this tool when they are making tests.

CHAPTER 7

Conclusion & Perspectives

Contents

1 Summary

Worldline is struggling to attract client projects. For this, the company needs to ensure success of its projects. One transversal team of the company, whose goal is to provide tools, expertise, and support to the other company teams, has for mission to enhance the project quality. Therefore, this thesis has for goal to identify causes of project failure and find solution to avoid them.

Chapter 2 first presented a study to check whether software metrics can be linked to project failure. Through literature study, mining of project data, and interviews of project leaders, we showed that metrics linked to success cannot be found. All these studies intervene *a posteriori* on projects. Consequently, it seems impossible for a new project to identify which metric or set of metrics could be used to assess success. Thanks to this study, we were able to extract some topics that are of interest for the project leaders of Worldline: communication, external software frameworks, software quality, and tests.

To check these results to more Worldline developers, we conducted another survey on 131 projects members of Worldline to obtain their insights on project quality. As a conclusion, it seems that selecting automatic tests after a code change can help project members to improve their project.

These studies are in Blondeau et al. [2015a,b].

Test selection approaches have to be designed for Worldline environment by respecting the constraints of the language and frameworks they use. Second, the testing behavior of Worldline developers have to be known to ensure a good adaptation of the tool to their practices.

Chapter 3 studies the state of the art. First, it showed that control flow graph approaches for test selection answer our need for Worldline. Among these approaches, two types exist: the static and the dynamic one. Literature does not conclude on the prevalence of the one above the other. Second, tooling for test selection has to satisfy the Worldline environment, existing tools are not tailored to it. Third, while studying developers behaviors, literature mainly implies open-source applications or student, which may not apply to Worldline. However, methodologies presented in it are interesting and are used as basis for comparison.

Chapter 4 presents the issues encountered by test selection approaches due to the particularities of the Worldline environment, the methodologies to resolve them, and the impact of their resolution on the test selection approach. We concluded that: First, the issues we discovered might be intertwined which means that a given selection method can exhibit several of the issues we identified. As a consequence, solving only one of the issues would not allow to recover all the tests that cover this method. However, by solving several issues at once, we are able to fully resolve any of the issues. Second, problems interact differently on the projects. Each project follows its own guidelines and uses its own frameworks that can influence the test selection. But test selection based on static approach gives satisfying results. This work is described in Blondeau et al. [2015c, 2016a,b].

Chapter 5 describes the experiment about testing habits of Worldline's developers. It gives results on their usage. This empirical study was inspired by two previous case studies [Beller et al., 2015, Gligoric et al., 2014]. Our conclusions are:

- Test practice in the company is better than what is described in Beller et al. [2015] with open-source, students, and industrials. It was an unexpected conclusion compared to the one of Chapter 2, where one root cause of project failure was the bypass of the qualification tests. But, this conclusion could be biased by the voluntary participation to the experiment;
- Developers do perform test selection, mostly reducing the test suite to one test (more than half of the test sessions ran only one test);
- Manual test selection is not accurate, many impacted tests are not launched after a change (recall=29%) and others are whereas they did not need to (precision=37%). We noted a tendency to err on the side of minimality rather than safeness.
- Contrary to intuition, amount of test execution or test selection does not depend on the size of the test suite nor on the duration of the tests: Shorter tests are not executed more often, and the number of test runs is not impacted by the total number of tests available in a project. This is good news at it reinforces the need to provide adequate test selection mechanisms to help software developers getting faster and better feedback.

This study is described in length in Blondeau et al. [2017].

In **Chapter 6**, we tried to improve testing behavior of Worldline developers. For this, we developed a new tool allowing them to automatically select tests on their projects. Few Worldline developers installed the plugin. In light of the absence of data, we decided to focus on the qualitative part by performing interviews with the four developers that installed the test selection plugin. Their opinions are mixed about the test selection tool: On the one hand, they spent less time to find the tests they should relaunch after a modification in their source code. But, on the other hand, the test selection tool slows three developers on four in their daily tasks when tests executions are performed at inappropriate moments. Finally, the developers suggested some improvements: add some test selection strategies and improve the computing performance of the tests to relaunch. But, they are all ready to continue to use this tool when they are making tests.

2 Contributions

The main contributions of this thesis are:

- An audit about software quality and root causes of project health. Developers were interviewed and surveyed to fulfill this goal.
- A classification of problems that could appear when trying to identify the tests related to a method change. Concrete examples of these problems and possible solutions are listed. These problems were identified from Worldline projects but remain relevant for Object Oriented projects in general, and Java ones in particular.
- A field study on how Worldline developers launch tests in their daily practice, whether they select tests and why they do it. This study in industrial context has been compared to ones on open source projects. Moreover, results are reinforced by interviews with developers involved in the study.
- A test selection tool adapted to the environment of Worldline. This tool can be used in closed contexts, *i.e.,* IT companies using Java on huge projects. This tool has also been replicated in the Pharo development environment (see Section 3.2).

3 Future Work

We separate future work on industrial and academic points of view.

3.1 Industrial

From an industrial point of view, the results of the surveys and interviews of the developers gave a new glance on the practices of the company, regarding testing but

also on software quality in general. Some of the results will inspire the transversal team. The survey concludes that project failure could be decreased by a better understanding of the specification by the client, an improvement of the communication between the project team and the client, and, inside the project team.

Nowadays, Agile methodologies try to solve these communication issues by setting up regular meetings, by adding a frame between the client and the team, and by giving directives to manage the development tasks inside the team. So, the Agile approach seems adapted to the problems encountered by the Worldline developers. The transversal team has now arguments to justify such an approach.

The test selection plugin was not really used by the developers of Worldline. We think that the critical mass of users was not reached to establish a momentum. By convincing more developers to install the plugin, and to use it, we hope to get interesting results and make developers win time in the long term. Effectively, the ones that installed it decided to continue to use it during their developments. It gives them a visual feedback on the tests that they should relaunch and forces them to think about testing during the whole development of the application. They are more confident in their code and, with a complete team using our test selection plugin, they would be more confident in the code written by the others. However, some features are lacking and the plugin can be improved to better match the requirements of the developer teams. Moreover, other maintainers of the plugin should be trained. The goal is that the knowledge of test selection keeps living inside Worldline.

Our work was presented to the Worldline's testing community. Despite the fact that there were few participants to the experiment, Worldline testers have been introduced to test selection. Interviews and surveys have shown that this topic is significant for the developers in terms of software quality. Liveliness around this topic have to be pursued.

By interviewing developers and analyzing our data, we saw that developers were not launching tests every day. Actually, at Worldline, most of the tests are made manually and developers loose time executing them. Automatic tests could avoid this loss. Therefore, initiatives have been launched to automatize the tests, *i.e.,* to implement the manual tests into automatic ones. But, it is a long term issue because the test automation debt is substantial. To add faster automated tests on the projects, an idea is to generate automatically the tests. As soon as a developer changes an uncovered piece of code, one can suggest him to generate automatically a test covering the change. It will improve the coverage of the application.

3.2 Academic

From an academic point of view, the studies on the developers made in Chapter 5 were extended to another environment than the Worldline one: the Pharo commu-

nity. In this open source community, developers seem more prone to writing and launching tests during the development. Pharo is a live environment: all the objects can be modified by the user and provide an immediate result. It gives the possibility to the user to break easily any part of the application. Moreover, the language is dynamically typed. It gives more freedom for the developers but it adds potential type matching issues. For these reasons, the need for testing is more than mandatory to keep the environment in a flawless state. We studied this environment which led to a research paper: Verhaeghe et al. [2017]. We conclude that Pharo developers run often their tests: they run twice more tests than in the Worldline study. They also practice test selection depending of the duration of the tests and select tests by their relevancy. Compared to Gligoric et al. [2014] and Beller et al. [2015] where the analysis where on various Java communities, we studied the behavior of the developers within another language, Pharo, in a smaller but uniform community.

Test selection may be a first step of the coevolution between the tests and the application source code. Test selection allows us able to match the tests with their source code and vice-versa when a piece of code is changed by the developer. If the associated tests are passing, the behavior of the application does not change and the modification is trustworthy (assuring that the tests cover well the code). Else, if the tests are failing, the change impacted negatively the application. Another change has to be made to reestablish the coherence of the application: either by modifying the test, or a part of the program. An approach for the coevolution of the source code and the tests could advise the developer on possible modifications to perform to make the tests green again. This approach would spare developers hours of investigation to discover the root causes of a bug.

APPENDIX A

Appendix

1 Analysis of Project Data

	Critical bugs	Major bugs	Minor bugs	Qualification bugs	Acceptance bugs	Total bugs	Production bugs	Δ Qualif. & Accept.	Realized budget	Predicted budget	Men*Days of slippage	Month in slippage	Is slippage	Intermediate releases	Project Name length
Critical bugs	1	0.82	0.74	0.6	0.62	0.89									
Major bugs	0.82	1	0.9	0.75	0.47	0.96	0.36	0.29							
Minor bugs	0.74	0.9	1	0.76	0.42	0.94	0.34	0.3							
Qualification bugs	0.6	0.75	0.76	1	0.58	0.74									
Acceptance bugs	0.62	0.47	0.42	0.58	1	0.55		-0.19							
Total bugs	0.89	0.96	0.94	0.74	0.55	1		0.21							
Production bugs		0.36	0.34				1								
Δ Qualif. & Accept.		0.29	0.3		-0.19	0.21		1							
Realized budget									1	0.97	0.41				
Predicted budget									0.97	1	0.26				
Men*Days of slippage									0.41	0.26	1				
Month in slippage												1	0.59	0.63	-0.25
Is slippage												0.59	1	0.48	
Intermediate releases												0.63	0.48	1	
Project Name length												-0.25			1

Figure A.1: Correlation matrix between each metric of the sample

Bibliography

Sven Amann, Sebastian Proksch, Sarah Nadi, and Mira Mezini. A Study of Visual Studio Usage in Practice. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 124–134. IEEE, 2016.
pp. 40 and 41.

Adrian Bachmann and Abraham Bernstein. Software process data quality and characteristics: A historical view on open and closed source projects. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, IWPSE-Evol '09, pages 119–128, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-678-6. doi: 10.1145/1595808.1595830. URL http://doi.acm.org/10.1145/1595808.1595830.

Linda Badri, Mourad Badri, and Daniel St-Yves. Supporting predictive change impact analysis: a control call graph based technique. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference*, APSEC'05, pages 9–15. IEEE, dec 2005. doi: 10.1109/APSEC.2005.100.
pp. 35 and 36.

Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, How, and Why Developers (Do Not) Test in Their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 179–190, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786843. URL http://doi.acm.org/10.1145/2786805.2786843.
pp. xiii, 40, 42, 43, 75, 76, 77, 78, 82, 83, 84, 85, 86, 87, 88, 91, 92, 93, 110, and 113.

Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *Proceedings of Future of Software Engineering (FOSE'07) at 29th International Conference on Software Engineering*, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2829-5. doi: 10.1109/FOSE.2007.25.
pp. 2.

A. Beszedes, T. Gergely, L. Schrettner, J. Jasz, L. Lango, and T. Gyimothy. Code Coverage-based Regression Test Selection and Prioritization in WebKit. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages

46–55, sep 2012. doi: 10.1109/ICSM.2012.6405252.
pp. 34.

Muhammad U. Bhatti, Nicolas Anquetil, and Stéphane Ducasse. An environment for dedicated software analysis tools. *ERCIM News*, 88:12–13, January 2012. URL http://ercim-news.ercim.eu/images/stories/EN88/EN88-web.pdf.
pp. 13.

Swarnendu Biswas, Rajib Mall, Manoranjan Satpathy, and Srihari Sukumaran. Regression test selection techniques: A survey. *Informatica (03505596)*, 35(3), 2011.
pp. 33, 46, 58, and 69.

Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009. ISBN 978-3-9523341-4-0. URL http://rmod.inria.fr/archives/books/Blac09a-PBE1-2013-07-29.pdf.
pp. 13.

Vincent Blondeau, Nicolas Anquetil, Stéphane Ducasse, Sylvain Cresson, and Pascal Croisy. Software metrics to predict the health of a project? In *IWST'15*, Brescia, Italy, July 2015a. doi: 10.1145/2811237.2811294. URL http://rmod.inria.fr/archives/papers/Blon15a-IWST-Software_metrics_to_predict_the_health_of_a_project.pdf.
pp. 109.

Vincent Blondeau, Sylvain Cresson, Pascal Croisy, Anne Etien, Nicolas Anquetil, and Stéphane Ducasse. Predicting the Health of a Project? An Assessment in a Major IT Company. In *SATToSE'15*, Mons, Belgium, July 2015b. URL http://rmod.inria.fr/archives/papers/Blon15b-SATToSE-Predicting_the_Health_of_a_Project-An_Assessment_in_a_Major_IT_company.pdf.
pp. 109.

Vincent Blondeau, Sylvain Cresson, Pascal Croisy, Anne Etien, Nicolas Anquetil, and Stéphane Ducasse. Dynamic and Static Approaches Comparison for Test Suite Reduction in Industry. In *BENEVOL'15: 14th BElgian-NEtherlands software eVOLution seminar*, Lille, France, December 2015c. URL http://rmod.inria.fr/archives/papers/Blon15c-BENEVOL-Blondeau_Vincent-Dynamic_and_Static_Approaches_Comparison_for_Test_Suite_Reduction_in_Industry.pdf.
pp. 110.

Vincent Blondeau, Nicolas Anquetil, Stéphane Ducasse, Sylvain Cresson, and Pascal Croisy. Test selection with moose in industry: Impact of granularity. In *International Workshop on Smalltalk Technologies IWST'16*, Prague, Czech Republic, September 2016a. URL http://rmod.inria.fr/archives/papers/Blon16b-IWST-Test_

Selection_with_Moose_In_Industry.pdf.
pp. 110.

Vincent Blondeau, Anne Etien, Nicolas Anquetil, Sylvain Cresson, Pascal Croisy, and Stéphane Ducasse. Test case selection in industry: An analysis of issues related to static approaches. *Software Quality Journal*, pages 1–35, 2016b. ISSN 1573-1367. doi: 10.1007/s11219-016-9328-4. URL http://rmod.inria.fr/archives/papers/Blon16a-Software_Quality_Journal-Test_Case_Selection_in_Industry-An_Analysis_of_Issues_Related_to_Static_Approaches.pdf.
pp. 110.

Vincent Blondeau, Anne Etien, Nicolas Anquetil, Sylvain Cresson, Pascal Croisy, and Stéphane Ducasse. What are the Testing Habits of Developers? A Case Study in a Large IT Company. In *Proceedings of the 21st IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*, Shanghai, China, August 2017. URL https://hal.inria.fr/hal-01571655/file/ICSME-FinalVersion-PDF_eXpress-Certified.pdf.
pp. 110.

Xia Cai, Michael R. Lyu, and Kam-Fai Wong. *A Generic Environment for COTS Testing and Quality Prediction*, pages 315–347. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-27071-3. doi: 10.1007/3-540-27071-X_15. URL https://doi.org/10.1007/3-540-27071-X_15.
pp. 10.

A. Capiluppi and J. Fernandez-Ramil. A model to predict anti-regressive efforts in open source software. In *23rd IEEE International Conference on Software Maintenance*, oct 2007. URL http://oro.open.ac.uk/8243/.

Andrea Capiluppi, Alvaro Faria, and J. F. Ramil. Exploring the relationship between cumulative change and complexity in an open source system evolution. In *Proceedings of the 9th European Conference on Software Maintenance and Re-engineering*, 2005. URL http://oro.open.ac.uk/12330/.

Narciso Cerpa, Matthew Bardeen, Barbara Kitchenham, and June Verner. Evaluating logistic regression models to estimate software project outcomes. *Information and Software Technology*, 52(9):934 – 944, 2010. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2010.03.011. URL http://www.sciencedirect.com/science/article/pii/S0950584910000595.

Narciso Cerpa, Matthew Bardeen, CÃ©sar A. Astudillo, and June Verner. Evaluating different families of prediction methods for estimating software project outcomes. *Journal of Systems and Software*, 112(Supplement C):48 – 64,

2016. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2015.10.011. URL http://www.sciencedirect.com/science/article/pii/S016412121500223X.
pp. 11.

Junya Debari, Osamu Mizuno, Tohru Kikuno, Nahomi Kikuchi, and Masayuki Hirayama. *On Deriving Actions for Improving Cost Overrun by Applying Association Rule Mining to Industrial Project Repository*, pages 51–62. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-79588-9. doi: 10.1007/978-3-540-79588-9_6. URL https://doi.org/10.1007/978-3-540-79588-9_6.
pp. 11.

Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems. In *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools*, CoSET '00, June 2000. URL http://scg.unibe.ch/archive/papers/Duca00bMooseCoset.pdf.
pp. 38.

Stéphane Ducasse, Nicolas Anquetil, Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, and Tudor Girba. MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family. Technical report, RMod – INRIA Lille-Nord Europe, 2011. URL http://rmod.inria.fr/archives/reports/Duca11c-Cutter-deliverable22-MSE-FAMIX30.pdf.
pp. 38.

Edward Dunn Ekelund and Emelie Engström. Efficient regression testing based on test history: An industrial evaluation. In *International Conference on Software Maintenance and Evolution*. IEEE Computer Society, 2015.
pp. 32, 34, 71, and 72.

S. Elbaum, P. Kallakuri, A. G. Malishevsky, G. Rothermel, and S. Kanduri. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. *Journal of Software Testing, Verification, and Reliability*, 2003.
pp. 35.

Emelie Engström, Mats Skoglund, and Per Runeson. Empirical Evaluations of Regression Test Selection Techniques: A Systematic Review. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 22–31. ACM, 2008.
pp. 31 and 33.

Emelie Engström, Per Runeson, and Mats Skoglund. A Systematic Review on Regression Test Selection Techniques. *Information and Software Technology*,

52(1):14–30, 2010.
pp. 31, 33, 34, 35, 59, and 69.

Michael D Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27. Citeseer, 2003.
pp. 31, 33, and 34.

Camilo Fitzgerald, Emmanuel Letier, and Anthony Finkelstein. Early failure prediction in feature request management systems: an extended study. *Requirements Engineering*, 17(2):117–132, Jun 2012. ISSN 1432-010X. doi: 10.1007/s00766-012-0150-7. URL https://doi.org/10.1007/s00766-012-0150-7.
pp. 9.

Takafumi Fukushima, Yasutaka Kamei, Shane McIntosh, Kazuhiro Yamashita, and Naoyasu Ubayashi. An empirical study of just-in-time defect prediction using cross-project models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 172–181, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2863-0. doi: 10.1145/2597073.2597075. URL http://doi.acm.org/10.1145/2597073.2597075.
pp. 9.

Milos Gligoric, Stas Negara, Owolabi Legunsen, and Darko Marinov. An Empirical Evaluation and Comparison of Manual and Automated Test Selection. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 361–372. ACM, 2014.
pp. xii, 40, 42, 43, 75, 76, 77, 79, 81, 83, 87, 89, 90, 91, 92, 93, 110, and 113.

Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 211–222, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3620-8. doi: 10.1145/2771783.2771784. URL http://doi.acm.org/10.1145/2771783.2771784.
pp. 35 and 36.

Maurice H. Halstead. *Elements of Software Science*. Elsevier North-Holland, 1977.

Will G Hopkins. *A new view of statistics*. Will G. Hopkins, 1997.
pp. 18.

Siw Elisabeth Hove and Bente Anda. Experiences from conducting semi-structured interviews in empirical software engineering research. In *Proceedings of the 11th IEEE International Software Metrics Symposium*, METRICS '05, pages 23–, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2371-4.

doi: 10.1109/METRICS.2005.24. URL http://dx.doi.org/10.1109/METRICS.2005.24.
pp. 80.

Pei Hsia, Xiaolin Li, David Chenho Kung, Chih-Tung Hsu, Liang Li, Yasufumi Toyoshima, and Cris Chen. A technique for the selective revalidation of oo software. *Journal of Software Maintenance: Research and Practice*, 9(4):217–233, 1997.
pp. 35.

V. Hurdugaci and A. Zaidman. Aiding software developers to maintain developer tests. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 11–20, mar 2012. doi: 10.1109/CSMR.2012.12.
pp. 36 and 107.

Judit Jász, Árpád Beszédes, Tibor Gyimóthy, and Václav Rajlich. Static execute after/before as a replacement of traditional software dependencies. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 137–146. IEEE, 2008.
pp. 35 and 36.

I.T. Jolliffe. *Principal Component Analysis*. Springer Verlag, 1986.
pp. 18.

Marian Jureczko and Lech Madeyski. Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, pages 9:1–9:10, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0404-7. doi: 10.1145/1868328.1868342. URL http://doi.acm.org/10.1145/1868328.1868342.
pp. 10.

Jussi Kasurinen, Ossi Taipale, and Kari Smolander. Software test automation in practice: empirical observations. *Advances in Software Engineering*, 2010, 2010.
pp. 40.

Rafaqut Kazmi, Dayang N. A. Jawawi, Radziah Mohamad, and Imran Ghani. Effective regression test case selection: A systematic literature review. *ACM Comput. Surv.*, 50(2):29:1–29:32, May 2017. ISSN 0360-0300. doi: 10.1145/3057269. URL http://doi.acm.org/10.1145/3057269.
pp. 31 and 32.

Staffs Keele et al. Guidelines for performing systematic literature reviews in software engineering. In *Technical report, Ver. 2.3 EBSE Technical Report. EBSE.*

sn, 2007.
pp. 8.

Stuart Kent. Model driven engineering. In *Integrated formal methods*, pages 286–298. Springer, 2002.
pp. 46.

Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. An extensive study of static regression test selection in modern software evolution. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 583–594. ACM, 2016.
pp. 34.

R. Lingampally, A. Gupta, and P. Jalote. A multipurpose code coverage tool for java. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 261b–261b, jan 2007. doi: 10.1109/HICSS.2007.24.
pp. 37.

Elton Mayo. *The human problems of an industrial civilization*. Macmillan Co., 1933.
pp. 92.

Gerard Meszaros. *XUnit Test Patterns – Refactoring Test Code*. Addison Wesley, June 2007.
pp. 2.

Osamu Mizuno, Takanari Hamasaki, Yasunari Takagi, and Tohru Kikuno. *An Empirical Evaluation of Predicting Runaway Software Projects Using Bayesian Classification*, pages 263–273. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-24659-6. doi: 10.1007/978-3-540-24659-6_19. URL https://doi.org/10.1007/978-3-540-24659-6_19.

Hussan Munir, Krzysztof Wnuk, Kai Petersen, and Misagh Moayyed. An experimental evaluation of test driven development vs. test-last development with industry professionals. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, page 50. ACM, 2014.
pp. 40.

Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 452–461, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. doi: 10.1145/1134285.1134349. URL

http://doi.acm.org/10.1145/1134285.1134349.
pp. 10.

Raphael Pham, Stephan Kiesling, Olga Liskin, Leif Singer, and Kurt Schneider. Enablers, inhibitors, and perceptions of testing in novice software teams. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 30–40. ACM, 2014.
pp. 40.

James Piggot and Chintan Amrit. *How Healthy Is My Project? Open Source Project Attributes as Indicators of Success*, pages 30–44. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-38928-3. doi: 10.1007/978-3-642-38928-3_3. URL https://doi.org/10.1007/978-3-642-38928-3_3.
pp. 11.

Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 33:1–33:11, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1614-9. doi: 10.1145/2393596.2393634. URL http://doi.acm.org/10.1145/2393596.2393634.
pp. 40, 41, and 57.

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013. URL http://www.R-project.org/.
pp. 13.

Jacek Ratzinger, Martin Pinzger, and Harald Gall. *EQ-Mine: Predicting Short-Term Defects for Software Evolution*, pages 12–26. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-71289-3. doi: 10.1007/978-3-540-71289-3_3. URL https://doi.org/10.1007/978-3-540-71289-3_3.
pp. 10.

Gregg Rothermel and Mary Jean Harrold. A Safe, Efficient Algorithm for Regression Test Selection. In *Proceedings of the International Conference on Software Maintenance (ICSM '93)*, pages 358–367. IEEE, September 1993.
pp. 31 and 34.

Per Runeson. A survey of unit testing practices. *IEEE software*, 23(4):22–29, 2006.
pp. 40.

Per Runeson and Martin Höst. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical software engineering*, 14

(2):131–164, 2009.
pp. 69 and 91.

Quinten David Soetens, Serge Demeyer, and Andy Zaidman. Change-based test selection in the presence of developer tests. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 101–110. IEEE, 2013.
pp. 36, 71, and 72.

Abbas Tashakkori and Charles Teddlie. *Mixed methodology: Combining qualitative and quantitative approaches*, volume 46. Sage, 1998.
pp. 80.

Dávid Tengeri, Ferenc Horváth, Árpád Beszédes, Tamás Gergely, and Tibor Gyimóthy. Negative effects of bytecode instrumentation on Java source code coverage. In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, pages 225–235, March 2016.
pp. 70 and 71.

Burak Turhan, Ayşe Tosun Mısırlı, and Ayşe Bener. Empirical evaluation of the effects of mixed project data on learning defect predictors. *Information and Software Technology*, 55(6):1101–1118, 2013.
pp. 10.

Benoit Verhaeghe, Nicolas Anquetil, Stéphane Ducasse, and Vincent Blondeau. Usage of tests in an open-source community. In *IWST'17*, Maribor, Slovenia, September 2017. URL https://hal.inria.fr/hal-01579106.
pp. 113.

J.M. Verner, W.M. Evanco, and N. Cerpa. State of the practice: An exploratory analysis of schedule estimation and software project success prediction. *Information and Software Technology*, 49(2):181 – 193, 2007. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2006.05.001. URL http://www.sciencedirect.com/science/article/pii/S095058490600067X.
pp. 11.

L. White, K. Jaber, and B. Robinson. Utilization of extended firewall for object-oriented regression testing. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 695–698, sep 2005. doi: 10.1109/ICSM.2005.101.
pp. 35.

L.J. White and H.K.N. Leung. A firewall concept for both control-flow and data-flow in regression integration testing. In *Software Maintenance, 1992. Proceedings., Conference on*, pages 262–271, nov 1992. doi: 10.1109/ICSM.1992.242535.
pp. 34.

D. Willmor and S.M. Embury. A safe regression test selection technique for database-driven applications. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 421–430, sep 2005. doi: 10.1109/ICSM.2005.15.
pp. 34.

Claes Wohlin and Anneliese Amschler Andrews. *Evaluation of Three Methods to Predict Project Success: A Case Study*, pages 385–398. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-31640-4. doi: 10.1007/11497455_31. URL https://doi.org/10.1007/11497455_31.
pp. 11.

Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000. ISBN 0-7923-8682-5.
pp. 80.

Zhenchang Xing and Eleni Stroulia. UMLDiff: An Algorithm for Object-oriented Design Differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 54–65, New York, NY, USA, 2005. ACM. ISBN 1-58113-993-4. doi: 10.1145/1101908.1101919. URL http://doi.acm.org/10.1145/1101908.1101919.
pp. 100.

S. Yoo and M. Harman. Regression Testing Minimization, Selection and Prioritization: A Survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012. ISSN 1099-1689. doi: 10.1002/stvr.430. URL http://dx.doi.org/10.1002/stvr.430.
pp. 31.

Andy Zaidman, Bart Rompaey, Arie Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Softw. Engg.*, 16(3):325–364, June 2011. ISSN 1382-3256. doi: 10.1007/s10664-010-9143-7. URL http://dx.doi.org/10.1007/s10664-010-9143-7.
pp. 40 and 41.

J. Zheng, L. Williams, B. Robinson, and K. Smiley. Regression Test Selection for Black-box Dynamic Link Library Components. In *Incorporating COTS Software into Software Systems: Tools and Techniques, 2007. IWICSS '07. Second International Workshop on*, pages 9–9, may 2007. doi: 10.1109/IWICSS.2007.8.
pp. 35.

Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 91–100. ACM, 2009.
pp. 10, 11, and 39.